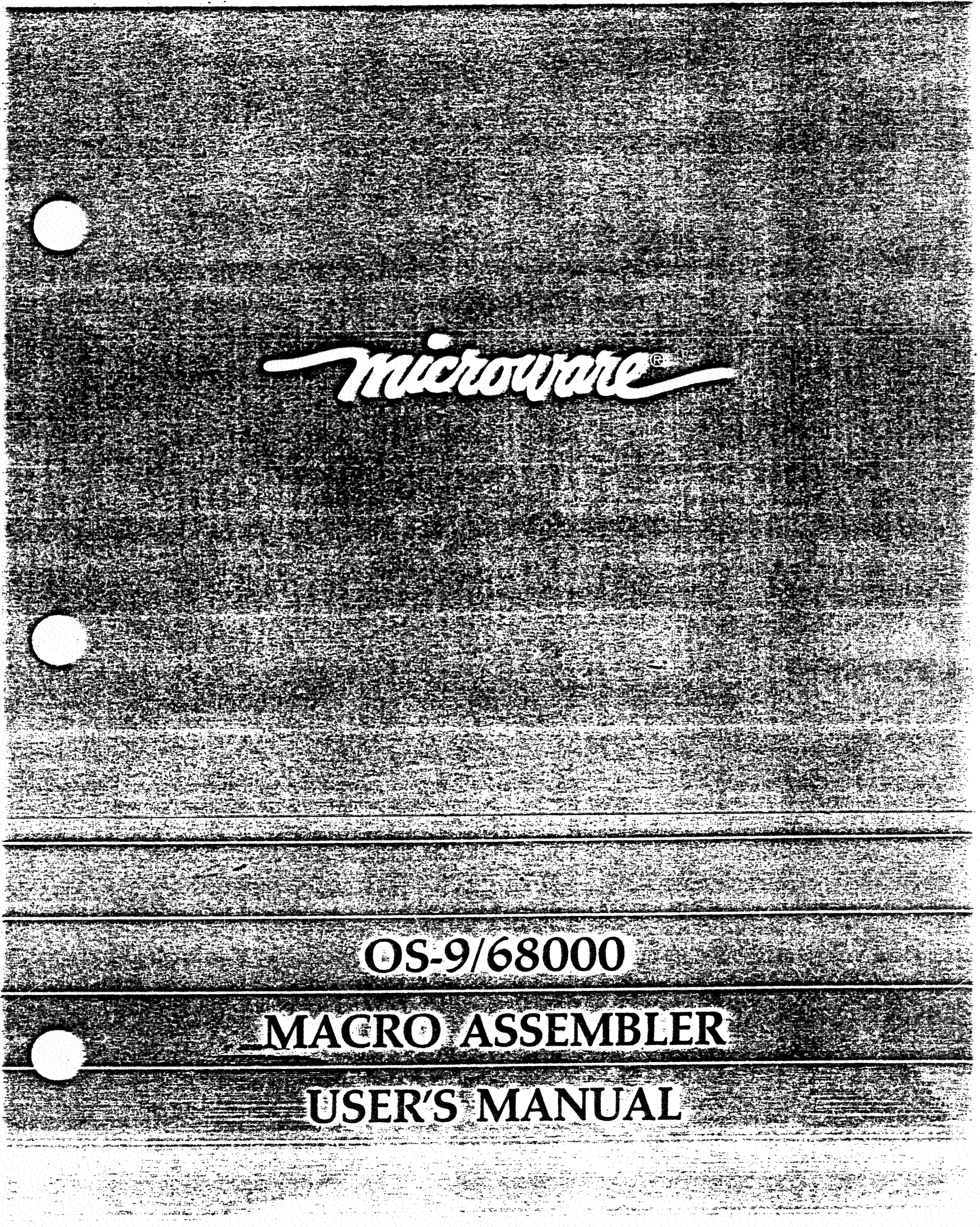


microware®

OS-9/68000

MACRO ASSEMBLER

USER'S MANUAL

OS-9/68000
MACRO ASSEMBLER
USER'S MANUAL

Copyright 1984 Microware Systems Corporation, All Rights Reserved. Reproduction of this document, in part or whole, by any means, electrical or otherwise, is prohibited except by written permission from Microware Systems Corporation.

The information contained herein is believed to be accurate as of the date of publication, however Microware will not be liable for any damages, including indirect or consequential, resulting from reliance upon the OS-9/68000 Macro Assembler Program or this documentation. The information contained herein is subject to change without notice.

OS-9 is a trademark of Microware System Corp. and Motorola Inc.
OS-9/68000 is a trademark of Microware Systems Corp.
UNIX is a trademark of Bell Laboratories

Revision D
Publication Date: October, 1985
Publication Editor: Walden Miller

Microware Systems Corporation
1866 N.W. 114th Street
Des Moines, Iowa 50322 U.S.A
Telephone 515-224-1929

INTRODUCTION

AN OVERVIEW

The Microware 68000 Macro Assembler is a full feature relocating macro assembler and linkage editor for OS-9/68000 systems. It was designed for use with hand-written or compiler-generated programs.

This software is available as a resident assembler for use on OS-9/68000 systems, as a cross-compiler for use on UNIX-based VAX or PDP-11 computers, or as a cross-compiler for OS-9 Level II-based 6809 computers.

Some of the main features of the assembler/linker package are:

1. Support for OS-9's modular, multi-tasking environment.
2. Built-in functions for calling OS-9 and generating system trap calls.
3. Supports use of position-independent, re-entrant code.
4. Allows programs to be written and assembled separately and then linked together which allows creation of standard subroutine libraries.
5. Full macro capabilities.
6. Can generate "stand alone" 68000 code

This manual describes how to use the Macro Assembler package and also discusses very basic programming techniques for the OS-9 environment. It is not intended to be a comprehensive course on 68000 assembly language programming. If you are not familiar with these topics, you should consult the Motorola 68000 programming manuals and one of the many assembly language programming books available at bookstores and libraries.

INSTALLATION

The distribution disk or tape contains a number of files that should be copied to the working system disk according to the accompanying instructions. The original distribution media should then be stored in a safe place for backup purposes.

The executable files for R68 and L68 should be copied to the system's execution directory. OS-9/68000 systems are generally supplied with these files already present in the "CMDS" directory.

For UNIX systems, new directories should be created for the cross-software on the root device: "/user" and "/user/bin" The macro assembler files should be then copied "/user/bin". Finally, "/user/bin" should be added to the Shell program search list so OS-9 cross file names do not conflict with other UNIX file names.

The DEFS and LIB directories contain include files that resolve system definitions.

On OS-9/68000 systems, the DEFS and LIB directories should be located on the root directory of the system default working disk device. On OS-9/6809 systems (for 68000 cross-compilation), the files in these directories should be called "/dd/DEFS.68K" and "/dd/LIB.68k".

On UNIX systems, these directories should be created within the "/user" directory discussed above (e.g., "/user/lib" and "/user/defs").

TABLE OF CONTENTS

Introduction i

Chapter 1 - Basic Information
- The Assember 1-1
- Assembly Language Program Development Process 1-2
- Running R68 1-2
- R68 Options 1-3
- Input File Format 1-4
  - Label Field 1-5
  - Operation Field 1-5
  - Operand Field 1-6
  - Comment Field 1-6
  - Assembly Listing Format 1-6
- Evaluation of Expressions 1-7
  - Expression Operands 1-7
  - Expression Operators 1-8
  - Expressions Involving External Symbols 1-9
  - Symbolic Names 1-9
- 68000 Assembly Language Mnemonics 1-10

Chapter 2 - Macros
- Introduction to Macros 2-1
- Macro Structure 2-2
- Macro Arguments 2-3
- Macro Automatic Internal Labels 2-5
- Additional Comments About Macros 2-6

Chapter 3 - Relocatable Program Sections
- Relocatable Program Sections 3-1
- Program Section Declarations: PSECT and VSECT 3-3
- Location Counters 3-5
- The Mainline Segment 3-5
- The PSECT Directive 3-6
- The VSECT Directive 3-8

Chapter 4 - Assembler Directive Statements
- Assembler Directive Statements 4-1
- END Statement 4-1
- EQU and SET Statements 4-2
- FAIL Statement 4-3
- IF, ELSE, and ENDC Statements 4-4
- NAM and TTL Statements 4-6
- OPT Statement 4-7
- PAG and SPC Statements 4-8
- REPT and ENDR Statements 4-9

OS-9/68000 MACRO ASSEMBLER USER'S MANUAL

Chapter 4 - Assembler Directive Statements (continued)

DS Statement - Define Storage 4-10
USE Statement 4-11

Chapter 5 - Pseudo-Instructions

Pseudo-Instructions 5-1
Align Statement 5-1
DC Statement 5-2
DZ Statement - Reserve Zero Bytes 5-3
DO, LO, ORG - Define Offset, Link Offset 5-4
OS9 Statement 5-5
Tcall Statement 5-6

Chapter 6 - The Linker

The Linker 6-1
The Root Psect 6-1
    The Subroutine Psect 6-2
Linker Execution 6-2
Linker Library Files 6-4
Linker Defined and Linker Recognized Symbols 6-5
The Linker Command Line 6-5
Linker Command Line Options 6-6
Linking Code For Non-OS-9 Systems 6-8

Chapter 7 - OS-9 Programming Techniques

OS-9 Programming Techniques 7-1
Program and Data Memory References 7-2
Data Area References 7-3
Code Area References 7-4

Appendix A - Example Program

Assembly Language Programming Example A-1

Appendix B - Error Messages

Error Messages B-1

Appendix C - ROF File Format

Relocatable Object File Format C-1
Header Section C-1
External Definition Section C-3
External Reference Section C-4
Local Reference Section C-4

Index

BASIC INFORMATION

THE ASSEMBLER

The assembler (called R68) permits sections of assembly language source programs to be independently translated to "relocatable object files" (ROFs). Global and local variables and program statement labels can be declared or referenced in each source program section. The assembler's macro facilities permit commonly used statement sequences to be defined, then used freely within the program with appropriate parameter substitution. R68 also supports conditional assembly and inclusion of library source files.

R68 is a two-pass assembler. During the first pass through the source program, the symbol table is created by scanning each line in order to identify symbolic name definitions. During the second pass, machine language instructions and data values are placed in the relocatable object file. The linker combines previously assembled relocatable object files in a separate pass.

The linkage editor (called L68) takes any number of program sections and/or library sections and combines them into a single executable OS-9 program. Global data and program references are automatically resolved during the linking process. The output of the linker is a binary executable file in the standard OS-9 memory module format. The linker also generates the appropriate "module header" for the program. For detailed information about memory modules refer to the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL".

A program which can be used to describe the contents of library files is also included in the package. The resident version of the program is called **"rdump"**. The cross-version is called **"rdump68"**. The syntax for rdump is:

```
rdump {<file>} [<opts>]
```

The <file> is a relocatable object file library that will usually have a suffix of ".r" or ".l".

The rdump options are:

-a	Displays information of all available options.
-g	Displays the global symbols defined.
-o	Displays the local relocation information.
-r	Displays the external references.

THE ASSEMBLY LANGUAGE PROGRAM DEVELOPMENT PROCESS

Writing and testing of assembly language programs using R68 and L68 involves a basic edit/assemble/link/test cycle. The assembler and linker can simplify this process because programs can be written in sections that can be assembled separately, then linked together to form the entire program.

In the event one program section must be changed for any reason, only the revised section has to be reassembled.

Here is a summary of the steps involved in the assembly language development process:

1. Create a source program file using the text editor.
2. Run the assembler (R68) to translate the source file(s) to a relocatable object module(s).
3. Use the text editor to correct any errors reported by R68 in the offending source file. Then go to step 2.
4. Combine all required relocatable modules using the linker (L68). If the linker reports errors, correct them and go to step 2.
5. Run and test the program. The OS-9 system debugger or user debugger (DEBUG) is frequently used to test programs.
6. If the program has bugs, use the text editor to correct the source file, then go to step 2.

RUNNING R68

R68 is a command program that can be run from the Shell, from a Shell procedure file, etc. The file and memory module names are "R68". The basic format of a command line to run the assembler is:

```
R68 filename [option(s)]  [>listing]
```

The only items absolutely required are the "R68" command name, and "filename" which is the source text file name. A following typical command line. They are functionally identical, but the second command uses an alternative way of combining options:

```
R68 prog5  -l -s --c >/p

R68 prog5 -ls --c >/p
```

In this example, the source program is read from the file "prog5". The source file name can be followed by an option list. This allows you to control various factors such as object file or listing generation, listing format control, etc.

An option list contains one or more options separated by spaces or commas. An option is turned on by its presence in the list preceded by a minus sign. A double minus sign followed by an option acts to turn the function off. In the example above, the options "l" and "s" are turned on, and "c" is turned off. If an option is not expressly given, the assembler will assume a default condition for it. Some command line options can be overridden by OPT statements within the source program (see the OPT statement description).

R68 automatically handles memory allocation for its working data area. Most of the data area memory is needed for the symbol table. R68 will request memory as needed up to the maximum available memory.

The final item, ">listing", allows the program listing generated by the assembler (on the standard output path) to be optionally redirected to another pathlist, which may be an output device such as a printer, a disk file, or a pipe to another program. Output redirection is handled by the Shell and not the assembler itself. If I/O redirection is omitted from the command line, the output will appear on your terminal display. In the above example, the listing output was directed to a device called "/p", which is the name of the printer on most OS-9 systems.

R68 OPTIONS

Up to 10 options are allowed on the command line. Each option is specified by a single letter preceded by a "-" or "--". Use "-" to turn on an option and "--" to turn the option off. The recognized assembler options are:

-c	Suppresses listing of conditional assembly lines in an assembler listing. (Default on)
-d <num>	Sets the number of lines per page (for listing) to <num>. (Default 66) *
-e	Suppresses printing of errors. (Default off)
-f	Uses a form feed for page eject, instead of line feeds. Uses form feed for top of form. (Default off) *
-g	Lists all code bytes generated. (Default off) *

* These options do not make sense unless the "-l" option is also used.

-l	Writes a formatted assembler listing to standard output. If not used, only error messages are printed. (Default off)

-m=<num>	Specifies machine assembler to be used with:
	`num = 0 = 68000 - Default`
	`1 = 68010 - Allow 68010 mnemonics/modes`
	`2 = 68020 - Allow 68010 and 68020 mnemonics`

-n	Omits line numbers from the assembler listing. This allows more room for comments. *
-o=<path>	Writes the relocatable output to the specified file (must be a mass storage file). (Default off)
-q	Suppresses warnings and nonfatal messages (quiet mode). (Default off)
-s	Prints the symbol table at the end of an assembly listing. (Default off)
-x	Suppresses macro expansion in assembler listing. (Default on) *

* These options do not make sense unless the "-l" option is also used.

INPUT FILE FORMAT

The R68 reads its input from an input file (path) which contains variable length lines of ASCII characters.

Each input line is a text string terminated by an end-of-line (return) character. The maximum length of the input line is 256 characters. Each line contains assembler statements as detailed in this manual. The line can have from one to four fields (fields are separated by spaces and/or tabs):

1. A label field (optional).
2. An operation field.
3. An operand field (contains 0 or more operands depending on the operation).
4. A comment field (optional).

NOTE: There are also two special cases:

1. An "*" (asterisk) in the first character position indicates a comment line. The entire line is printed in the listing, but is not otherwise processed.
2. Blank lines are also included in the listing, but are likewise ignored.

Label Fields

The label field begins in the first character position of the line. Labels are required by some statements (i.e., EQU and SET). Labels are not allowed on others (assembler directives such as SPC, TTL, etc.).

The first character of the line must be a space or tab if the line does not contain a label. If the label is present, the assembler defines it as the address of the first byte of where the instruction's object code will be assigned. An exception to the rule is that labels on SET and EQU statements are given the value of the result of evaluation of the operand field.

When a symbolic name in the label field of a source statement is followed by a ":" (colon), the name is known GLOBALLY by all modules that are linked together. Since the label is known globally, a branch or jump can be done to a location in another module. For a global variable, the data offset can be referred to by other modules.

If no colon appears after the label, the label will be known only in the PSECT where it is defined. Care must be taken to access the labels in the appropriate context.

The label must be a legal symbolic name consisting of from one to nine uppercase or lowercase characters, decimal digits, or the characters "$", "." or "_". The first character may not be a dollar sign, period or digit. Upper and lower case characters are distinct.

Labels (and names in general) must be unique. They can not be defined more than once in a program (except when used with the "SET" directive). Labels on SET and EQU statements are given the value of the result of evaluation of the operand field. These statements allow any value to be associated with a symbolic name.

The Operation Field

The Operation field specifies the machine language instruction or assembler directive statement mnemonic name. It immediately follows the label field. It is separated from the prior field by one or more spaces. R68 accepts instruction mnemonic names in either uppercase or lowercase characters.

Instructions cause two to ten bytes of object code to be generated depending on the specific instruction and addressing mode. Some assembler directive statements (such as dz and dc) also cause object code to be generated.

Many 68000 instructions require a size attribute such as "move.b d0,d1" or "move.w d1,(sp)". The default size is ".w" (word) if no size attribute is specified. For example, "move d0,d1" is a word move. Some instructions, however, have no choice of size attribute. In this case no size attribute is allowed.

Operand Field

The operand field follows the instruction field. They must be separated by at least one space or tab. Some instructions don't use an operand field. Other instructions and assembler directives require one to specify an addressing mode, operand address, and/or parameters. Some require a source operand and a destination operand, etc.

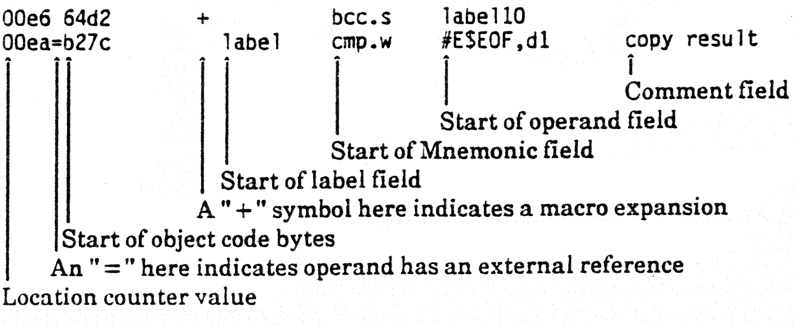
The sections describing the instructions and assembler directives explain the format for operand(s), if any.

Comment Field

The last field of the source statement is the optional comment field which can be used to include a descriptive comment in the source statement. This field is not processed other than being copied to the program listing.

Assembly Listing Format

If the "-l" option is given in the R68 command line, a formatted assembly listing will be written to the standard output path. The output listing has the following format:



```
00e6 64d2      +       bcc.s     label10
00ea=b27c        label cmp.w     #E$EOF,d1       copy result
```

Comment field
Start of operand field
Start of Mnemonic field
Start of label field
A "+" symbol here indicates a macro expansion
Start of object code bytes
An "=" here indicates operand has an external reference
Location counter value

BASIC INFORMATION

EVALUATION OF EXPRESSIONS

Operands of many instructions and assembler directives can include numeric expressions in one or more places. The assembler can evaluate expressions of almost any complexity using a form similar to the algebraic notation used in programming languages such as BASIC and FORTRAN.

Expressions consist of **operands**, which are symbolic names or constants, and **operators**, which specify an arithmetic or logical function. All assembler arithmetic uses long word (internally, 32 bit binary) signed or unsigned integers in the range of 0 to 4294967295 for unsigned numbers, or -2147483648 to +2147483647 for signed numbers.

In some cases, expressions must evaluate to a value which must fit in one byte (such as 8-bit displacement in branch instructions) and therefore must be in the range of 0 to 255 for unsigned values and -128 to 127 for signed values. In these cases, if the result of an expression is outside of this range an error message will be given. Similarly, instructions that require a 16 bit value must evaluate within the range of 0 - 65535 (unsigned) or -32768 to +32767 (signed).

Expressions are evaluated from left-to-right using the algebraic order of operations (i.e. multiplications and divisions are performed before additions and subtractions). Parentheses can be used to alter the natural order of evaluation.

Expression Operands

The following items may be used as operands within an expression:

DECIMAL NUMBERS: an optional minus sign followed by one to twelve digits. For example:

```
100         3164765
-32767      -999999
0           12
```

HEXADECIMAL NUMBERS: a dollar sign ("$" or "0x") followed by one to eight hexadecimal characters (0-9, A-F or a-f). For example:

```
$EC00       $1000
$3          $0300
0xFFFF      0xDEADFACE
```

BINARY NUMBERS: a percent sign ("%") followed by one to sixteen binary digits (0 or 1). For example:

```
%0101
%1111000011110000
%10101010
```

CHARACTER CONSTANTS: a single character enclosed by single quotes ("'"). For example:

```
'X'
'c'
'5'
```

SYMBOLIC NAMES: one to nine characters consisting of:

```
upper or lower case letters (A-Z, a-z)
digits (0-9)
special characters:  "_" (underscore)
                     "." (period)
                     "@" (at sign)
                     "$" (dollar sign)
```

The first character can not be a digit, a dollar sign or a period.

LOCATION COUNTER SYMBOL: the asterisk ("*") represents the current location counter value as of the beginning of the line.

Expression Operators

Operators used in expressions operate on one operand (negative and not) or on two operands (all others). The following table shows the available operators, listed in the order they are evaluated relative to each other, i.e, logical OR operations are performed before multiplications. Operators listed on the same line have identical precedence and are processed from left to right when they occur in the same expression.

Assembler Operators By Order of Evaluation	
- negative	^ logical NOT
& logical AND	! | logical OR
* multiplication	/ division
+ addition	- subtraction
<< shift left	>> shift right

Logical operations are performed bitwise, i.e., the logical function is performed bit-by-bit on each bit of the operands. Division and multiplication functions assume unsigned operands, but subtraction and addition work on signed (2's complement) or unsigned numbers. Division by zero or multiplication resulting in a product larger than 4294967295 have undefined results and are reported as errors.

Expressions Involving External Symbols

An external symbol is a symbol whose value is not known at the time the program section is assembled. The actual values of external references must be inserted later when the program is linked.

The linker can resolve a limited number of expressions involving external references. These expressions can consist only of simple addition and subtraction operations involving two operands at most. The following expression forms involving external references are supported. All other forms are illegal.

```
External + Absolute
External - Absolute
External - External
```

The linker performs subtraction by negating one operand and then adding it to the other operand. This method can cause problems on signed values of either word or byte length as the linker may report over/underflow errors. Therefore, care should be taken to minimize the complexity of expressions involving external names.

Symbolic Names

A symbolic name consists of from one to nine uppercase or lowercase characters, decimal digits, or the characters "$", "_", "." and "@". However, the first character can not be a digit, dollar sign or a period. The following are examples of legal symbol names:

```
HERE        there       SPL030      PGM_A
Q1020.1     t$integer   L.123.X     a002@
```

One thing to keep in mind is that the R68 does not match lowercase letters to uppercase letters. The names "val_A" and "VAL_A" are considered different names.

These are examples of some illegal symbol names with reasons why they are such:

`2move`	- starts with a digit
`main.backup`	- more than 9 characters - truncated to 9
`lbl#123`	- # is not a legal name character

Names are defined when first used as a label on an instruction or directive statement. They must be defined exactly one time in the program (except SET labels: see SET statement description). If a name is redefined (used as a label more than once) an error message is printed on subsequent definition(s).

If a symbolic name is used in an expression and has not been defined, the name is assumed to be external to the PSECT. Information will be recorded about the reference so the linker can adjust the operand accordingly. However, external names can not appear in operand expressions for assembler directives.

68000 ASSEMBLY LANGUAGE MNEMONICS

The assembler uses Motorola standard assembly language mnemonics and syntax. For more specific information about individual instructions, consult the book titled "M68000 16/32 Bit Microprocessor Programmer's Manual", Prentice-Hall, Fourth Edition, Motorola part number M68000UM(AD4).

The following register names are reserved and cannot be redefined or used out of context.

An - address register n
Dn - data register n
pc or pcr - Program counter
sr - status register
ccr - condition codes
ssp - supervisor stack pointer
usp - user stack pointer
sfc - source function code (68010, 68020)
dfc - destination function (68010, 68020)
Rn - data or address register n

The following information describes addressing mode syntax.

Dn	data register direct
An	address register direct
(An)	register indirect
(An)+	postincrement register indirect
-(An)	predecrement register indirect
d(An)	register indirect with offset
d(An,Rn.s)	Indexed register indirect with offset
xxx.w	absolute short
xxx	absolute long
d(pc)	pc relative with offset
d(pc,Rn.s)	pc relative with index and offset
#xxx	immediate data

BASIC INFORMATION

The following instruction mnemonic summary uses these conventions:

`<data>`	immediate data of appropriate size.
`.s`	means ".w", ".l", or ".b" The default is ".w" if the size is not explicitly given. Short branches and bsr use the ".s" literally.
`<ea>`	any legal addressing mode for the instruction.

MNEMONIC	DESCRIPTION
abcd Dy,Dx	Add decimal with Extend Register
abcd -(Ay),-(Ax)	Add decimal with Extend Memory
add.s <ea>,Dn	Add binary register
add.s Dn,<ea>	Add binary memory
adda.s <ea>,An	Add address (.w or .l only)
addi.s #<data>,<ea>	Add immediate
addq.s #<data>,<ea>	Add quick
addx.s Dy,Dx	Add extended register
addx.s -(Ay),-(Ax)	Add extended memory
and.s <ea>,Dn	And logical register
and.s Dn,<ea>	And logical memory
andi.s #<data>,<ea>	And immediate
andi #<data>,ccr	And immediate to cond code
andi #<data>,sr	And immediate to status register
NOTE: In the following three instructions, "d" represents the shift direction: l (for left) or r (for right).	
asd.s Dx,Dy	Arithmetic Shift register
asd.s #<data>,Dy	Arithmetic Shift immediate register
asd <ea>	Arithmetic Shift memory

MNEMONIC	DESCRIPTION
NOTE: In the following two instructions, "cc" represents the branch condition code; see the Motorola Programming Manual.	
`bcc <label>`	Conditional branch long
`bcc.s <label>`	Conditional branch short
`bchg.s Dn,<ea>`	Test bit and change register (.b or .l)
`bchg.s #<data>,<ea>`	Test bit and change immediate (.b or .l)
`bclr.s Dn,<ea>`	Test bit and clear register (.b or .l)
`bclr.s #<data>,<ea>`	Test bit and clear immediate (.b or .l)
`bra <label>`	Branch long
`bra.s <label>`	Branch short
`bset.s Dn,<ea>`	Test bit and set register (.b or .l)
`bset.s #<data>,<ea>`	Test bit and set immediate (.b or .l)
`bsr.s <label>`	Branch subroutine
`bsr <label>`	Branch subroutine long
`btst.s Dn,<ea>`	Test bit register (.b or .l)
`btst.s #<data>,<ea>`	Test bit immediate (.b or .l)
`chk <ea>,Dn`	Check register against bounds
`clr.s <ea>`	Clear operand
`cmp.s <ea>,Dn`	Compare data register
`cmpa.s <ea>,An`	Compare address register
`cmpi.s #<data>,<ea>`	Compare immediate
`cmpm.s (Ay)+,(Ax)+`	Compare memory
NOTE: "cc" in the following instruction represents the branch condition code; see the Motorola Programming Manual.	
`dbcc dn,<label>`	Test condition, decrement and branch

MNEMONIC	DESCRIPTION
`divs <ea>,Dn`	Signed divide
`divu <ea>,Dn`	Unsigned divide
`eor.s Dn,<ea>`	Exclusive OR
`eori.s #<data>,<ea>`	Exclusive OR immediate
`eori #<date>,ccr`	Exclusive OR condition code
`eori #<data>,sr`	Exclusive OR status register
`exg Rx,Ry`	Exchange registers
`ext.s Dn`	Sign extend (.w or .l)
`jmp <ea>`	Jump
`jsr <ea>`	Jump to subroutine
`lea <ea>,An`	Load effective address
`link An, #<displacement>`	Link and allocate
NOTE: In the following three instructions "d" represents the shift direction: l (for left) or r (for right).	
`lsd.s Dx,Dy`	Logical shift data
`lsd.s #<data>,Dy`	Logical shift immediate
`lsd <ea>`	Logical shift memory
`move.s <ea>,<ea>`	Move from source to destination
`move ccr,<ea>`	Move from condition codes. This is not available on the 68000. It is available only on the 68010.
`move <ea>,ccr`	Move to condition codes
`move <ea>,sr`	Move to status register
`move sr,<ea>`	Move from status register. This is privileged on the 68010. Avoid this instruction in programs that are to execute in "user" mode.
`move usp,An`	Move from user stack pointer

MNEMONIC	DESCRIPTION
`move An,usp`	Move to user stack pointer
`movea.s <ea>,An`	Move address (.w or .l)
`movec Rc,Rn`	Move from control reg.(68010)
`movec Rn,Rc`	Move to control reg.(68010)
`movem.s <ea>,<reg list>`	Move multiple `<reg list>`: `rn` register `rx-ry` consecutive registers `/` register delimiter `#<expr>` register list mask Examples: `d0` d0 only `d0/d4/a5` d0,d4,a5 `d0-d7/a0-a5` d0 through d7, a0 through a5
`movep.s dx,d(Ay)`	Move peripheral data (.w or .l) from register to memory
`movep.s d(Ay),dx`	Move peripheral data (.w or .l) from memory to register
`moveq.l #<data>,dn`	Move quick
`moves.s Rn,<ea>`	Move to address space (68010)
`moves.s <ea>,Rn`	Move from address space (68010)
`muls <ea>,Dn`	Signed multiply
`mulu <ea>,Dn`	Unsigned multiply
`nbcd <ea>`	Negate decimal with extend
`neg.s <ea>`	Negate
`neg.s <ea>`	Negate with extend
`nop`	No operation
`not.s <ea>`	Logical complement

MNEMONIC	DESCRIPTION
`or.s <ea>,Dn`	Inclusive OR register
`or.d Dn,<ea>`	Inclusive OR memory
`ori.s #data>,<ea>`	Inclusive OR immediate
`ori #<data>,ccr`	Inclusive OR condition codes
`ori #<data>,sr`	Inclusive OR status register
`pea <ea>`	Push effective address
`reset`	Reset external devices
NOTE: In the following six instructions "d" represents the shift direction: l (for left) or r (for right).	
`rod.s Dx,Dy`	Rotate without extend reg
`rod.s #<data>,dy`	Rotate without extend immediate
`rod <ea>`	Rotate without extend memory
`roxd.s Dx,Dy`	Rotate with extend register
`roxd.s #<data>,dy`	Rotate with extend immediate
`roxd <ea>`	Rotate with extend memory
`rtd #<displacement>`	Return and deallocate (68010)
`rte`	Return from exception
`rtr`	Return and restore condition codes
`rts`	Return from subroutine
`sbcd Dy,Dx`	Subtract decimal w/extend reg
`sbcd -(Ay),-(Ax)`	Subtract decimal w/extend mem
NOTE: "cc" in the following instruction represents the branch condition code; see the Motorola Programming Manual.	
`scc <ea>`	Set according to cond. regs
`stop #<data>`	Load and stop

MNEMONIC	DESCRIPTION
`sub.s <ea>,Dn`	Subtract binary register
`sub.s Dn,<ea>`	Subtract binary memory
`suba.s <ea>,An`	Subtract address (.w or .l)
`subi.s #<data>,<ea>`	Subtract immediate
`subq.s #<data>,<ea>`	Subtract quick
`subx.s Dy,Dx`	Subtract with extend register
`swap Dn`	Swap register halves
`tas <ea>`	Test and set operand
`trap #<vector>`	Trap
`trapv`	Trap on overflow
`tst.s <ea>`	Test operand
`unlk An`	Unlink

end of chapter 1

MACROS

INTRODUCTION TO MACROS

Often identical or similar sequences of instructions may be repeated in different places in a program. Writing a sequence of instructions repeatedly can be tedious if it is long or must be used a number of times.

A macro is a definition of an instruction sequence that can be used numerous places within a program. The macro is given a name which is used similarly to any other instruction mnemonic. Whenever R68 encounters the name of a macro in the instruction field, it outputs all the instructions given in the macro definition. In effect, macros allow the programmer to create "new" machine language instructions.

For example, suppose a program frequently must perform left shifts. This two-instruction sequence can be defined as a macro, for example:

```
dasl    MACRO
        asl.l dl
        roxl.l d0
        ENDM
```

The "MACRO" and "ENDM" directives specify the beginning and the end of the macro definition, respectively. The label of the "macro" directive specifies the name of the macro, "dasl" in this example. Now the "new" instruction can be used in the program.

In the example above, when R68 encounters the "dasl" macro, it actually can output code for "asl" and "roxl". Normally, only the macro name is listed, but an R68 option can be used to cause all instructions of the "macro expansion" to be listed.

Macros should not be confused with subroutines although they are similar in some ways. Macros repetitively duplicate an "in line" code sequence every time they are used and allow some alteration of the instruction operands. Subroutines appear exactly once, and never change. They are called using special instructions (BSR, JSR, and RTS).

In those cases where macros and subroutines can be used interchangeably, macros usually produce longer but slightly faster programs. Subroutines produce shorter and slightly slower programs. Short macros (up to 6 bytes or so) will almost always be faster and shorter than subroutines because of the overhead of the BSR and RTS instructions needed.

MACRO STRUCTURE

A macro definition consists of three sections:

```
<name>  MACRO   * the macro header assigns a name to the macro *
        .
        .
        body    * the macro body contains the macro statements *
        .
        .
        ENDM    * the macro terminator indicates the end of the macro *
```

The macro name must be defined by the label given in the MACRO statement. The name can be any legal assembler label. It is possible to redefine the 68000 instructions themselves by defining macros having identical names. This gives R68 the capability to be used as a "cross-assembler" for non-68000 processors by definition and/or redefinition of the instruction set of the target CPU.

CAUTION: Redefinition of assembler directives such as "ds" can have unpredictable consequences.

The body of the macro can contain any number of legal R68 instructions or directive statements including references to previously defined macros. The last statement of a macro definition must be ENDM.

The text of macro definitions are stored on a temporary file created and maintained by R68. This file has a large (1K byte) buffer to minimize disk accesses. Therefore, programs that use more than 1K of macro storage space should be arranged so that short, frequently used macros are defined first so they are kept in the memory buffer instead of disk space.

Macro calls may be nested, that is, the body of a macro definition may contain a call to another macro. For example:

```
times2  MACRO
        mac1
        mac1
        ENDM
```

The macro above consists of the "mac1" macro used twice. The definition of a new macro within another is not permitted. Macro calls may be nested up to eight deep.

MACROS

MACRO ARGUMENTS

Arguments permit variations in the expansion of a macro. Arguments can be used to specify operands, register names, constants, variables, etc., in each occurrence of a macro.

A macro can have up to nine formal arguments in the operand fields. Each argument consists of a backslash character and the sequence number of the formal argument (\1, \2 ... \9). When the macro is expanded, each formal argument is replaced by the corresponding text string "actual argument" given in the macro call. Arguments can be used in any part of the operand field not in the instruction or label fields. Formal arguments can be used in any order and any number of times.

For example, the macro below performs the typical instruction sequence to an I$WritLn:

```
writ    MACRO
        moveq  #\1,d0       get path
        moveq  #\2,d1       number of chars to write
        lea    \3(a6),a0    get address of buffer
        os9 I$WritLn
        ENDM
```

This macro uses three arguments: "\1" for the path number; "\2" for the number of characters to write; and "\3" for the address of the buffer. When "writ" is referenced, each argument is replaced by the corresponding string given in the macro call, for example:

```
writ 1,2,Buf
```

The macro call above will be expanded to the code sequence:

```
moveq #1,d0
moveq #2.d1
lea   Buf(a6),a0
os9 I$Writln
```

If an argument string includes special characters such as backslashes or commas, the string must be enclosed in double quotes.

An argument may be declared null by omitting all or some arguments in the macro call. This makes the corresponding argument an empty string so no substitution occurs when it is referenced.

There are two special argument operators that can be useful in constructing more complex macros. They are:

`\Ln`	Returns the length of the actual argument n, in bytes.
`\#`	Returns the number of actual arguments passed in the given macro call.

These special operators are most commonly used in conjunction with R68's conditional assembly facilities to test the validity of arguments used in a macro call, or to change the way a macro works according to the actual arguments used. When macros are performing error checking they can report errors using the FAIL directive. Here is an example using the "create" macro given on the previous page but expanded for error checking:

```
writ    MACRO
        ifne \#-3          must have exactly 3 args
        FAIL create:       must have three arguments
        endc
        ifgt \L3-29        file name can be 1 - 29 chars
        FAIL create:       file name too long
        endc

          {macro code}

        ENDM
```

MACROS

MACRO AUTOMATIC INTERNAL LABELS

Sometimes it is necessary to use labels within a macro. Labels are specified by "\@". Each time the macro is called, a unique label will be generated to avoid multiple definition errors. Within the expanded code "\@" will take on the form "@xxxxx" (xxxxx = a decimal number between 00000 and 99999).

More than one label may be specified in a macro by the addition of an extra character(s). For example, if two different labels are required in a macro, they can be specified by "\@A" and "\@B". In the first expansion of the macro, the labels would be "@001A" and "@001B", and in the second expansion they would be "@002A" and "002B". The extra characters may be appended before the "\" or after the "@".

Here is an example of a macro that uses internal labels:

```
test    MACRO
  {macro code}
\@A
  {macro code}
\@B
```

The first expansion will be:

```
  {macro code}
@001A
  {macro code}
@001B
```

The second expansion will be:

```
  {macro code}
@002A
  {macro code}
@002B
```

ADDITIONAL COMMENTS ABOUT MACROS

Macros can be an important and useful programming tool that can significantly extend R68's capabilities. In addition to creating instruction sequences, they can also be used to create complex constant tables and data structures.

WARNING: When macros are used they should be carefully documented. Macros can impair the readability of a program if they are used indiscriminately and unnecessarily. This can make it extremely difficult to understand the program logic.

end of chapter 2

RELOCATABLE PROGRAM SECTIONS

RELOCATABLE PROGRAM SECTIONS

A primary purpose of R68 is to permit programs to be composed of different segments that can be assembled separately.

NOTE: To clarify the following discussion, "segments" are synonymous with "source files".

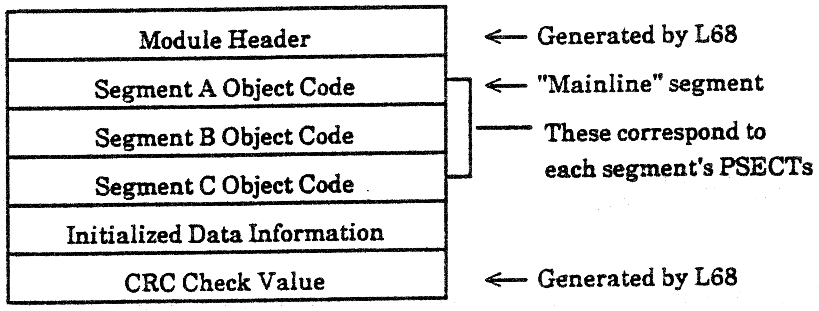
OS-9 processes use at least two separate areas of memory: the program object code in memory module format, and a data area used for the program's variables and the stack. The linker (L68) combines all of the segments into a single OS-9 memory module and a coordinated data storage area. By use of **global** symbolic names, code in each segment can reference variables declared in other segments, or may transfer program control to labels in other segments.

When the assembler source program for each segment is written, it must be divided into distinct **program sections** for variable storage definitions (VSECTs) and for program statements (PSECTs). The output of the assembler is a distinct **relocatable object file** (ROF) which contains the object code output plus information about the variable storage declarations for use by the linker.

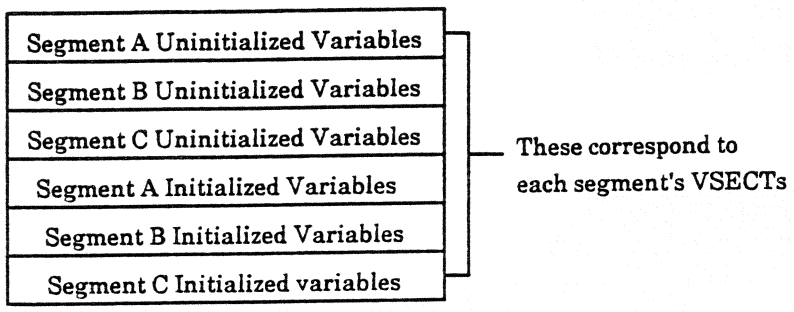
The linker reads the ROFs and assigns space in the data storage area and combines all the object code into a single executable memory module. As it does so, it must alter the operands of instructions to refer to the final variable assignments and must also adjust program control transfer instructions that refer to label in other segments.

For example, if three segments called "A", "B", and "C" are processed by the linker, the resulting memory allocation would be as shown in the simplified memory map shown on the next page.

Executable Memory Module


Module Header
Segment A Object Code
Segment B Object Code
Segment C Object Code
Initialized Data Information
CRC Check Value
Generated by L68
"Mainline" segment
These correspond to each segment's PSECTs
Generated by L68

Process Data Area


Segment A Uninitialized Variables
Segment B Uninitialized Variables
Segment C Uninitialized Variables
Segment A Initialized Variables
Segment B Initialized Variables
Segment C Initialized variables
These correspond to each segment's VSECTs

RELOCATABLE PROGRAM SECTIONS

PROGRAM SECTION DECLARATIONS: PSECT AND VSECT

Most program statements are included in sections called PSECTs (for "program sections") and VSECTs (for "variable sections").

The PSECT contains the program instructions and variable declarations. Each source file may have only one PSECT. Global symbols (i.e., labels with a ":" suffix) in this section are accessable from all other program segments. Similarly, statements in this section can reference global symbols properly defined in other program segments. The PSECT is terminated by a matching ENDS (or ENDSECT statement).

Global and local variable storage are declared inside one or more VSECTs within the PSECT. VSECTs usually are "nested" within a PSECT, but VSECTs can not be nested within themselves.

A variable declaration section begins with a VSECT statement and ends with a ENDS (or ENDSECT) statement. There are three types of variable declarations: initialized, uninitialized and remote. These correspond to the assembly language storage allocation mnemonics "dc" and "ds", respectively. L68 combines all initialized variable declarations from all program segments into a single initialized data memory area. Similarly, all the uninitialized data declarations are combined into a single uninitialized data memory area.

Certain types of statements can appear outside (usually before) the PSECT. These are generally SET, and EQU (and possibly the LO and DO statements). These are used to declare symbolic constants and symbolic offsets. Labels on these statements are local to the assembly of the source file and are not usable during assembly of other program segments. Additionally, these statements can not reference any global symbols.

For example, the "oskdefs" file is intended to be included outside the PSECT of each source program. Although technically a VSECT can similarly appear outside the PSECT, the usefulness of such a VSECT is limited to defining the expected type of an external symbol as a data area symbol because no actual storage would be assigned to it by the linker.

Diagram of Typical Program Layout

Description — **Example source**

```
     nam example
     ttl sections and declarations
```

local constant definitions

```
labconst equ 1
space equ $20
mode set 1
```

include local defs. file

```
     use /h0/defs/oskdefs
```

start of PSECT

```
     psect nam,typ,rev,ed,stack,entry
```

start of nested VSECT
global uninitialized data
global initialized data
local uninitialized data
local initialized data
end of VSECT

```
     vsect
gblunin: ds.l 1
gblinit: dc.b "string",0
locunin  ds.l 1
locinit  dc.b "hello",0
     ends
```

global code label
global uninitialized data reference
global initialized data reference
internal code reference
external code reference
external data reference

```
gblcode:
     move.l gblunin(a6),d0
     lea gblinit(a6),a0
     bsr.s intcode
     bsr extcode
     move.l d0,extdat(a6)
     rts
```

start of nested VSECT
local uninitialized data
end of VSECT

```
     vsect
indat ds.l 1
     ends
```

local code label

```
intcode rts
```

end of PSECT

```
     ends
```

NOTE: See Appendix A for a complete functional 68000 assembly language program example.

RELOCATABLE PROGRAM SECTIONS

LOCATION COUNTERS

R68 maintains a set of address counters that keep track of relative memory addresses of object code, initialized data, uninitialized data and remote data. It is important to remember that location counter values are relative and not the actual physical memory addresses. Actual memory locations are not known until the program has been linked and loaded into memory.

The PSECT statement resets the instruction and data location counters, and assembles subsequent instructions into the ROF object code file. As object code is generated, the instruction location counter is advanced accordingly. A "*" symbol can be used in expressions to refer to the current relative value of this counter.

VSECT causes R68 to use the variable (data) location counters and places information about subsequently declared variables into the appropriate ROF data description area. As variables are declared the **initialized data location counter** and the **uninitialized data location counter** are advanced accordingly. "*" represents the value of the initialized data location. The uninitialized data counter is not directly accessable.

There is no way to preset any of the counters to a specific value (i.e., there is no "ORG" statement for the data or instruction counters).

THE MAINLINE SEGMENT

Each complete program must have one segment which is called the **mainline segment**. It is special because it gives the linker the information necessary to create the OS-9 module header such as the module name, the initial entry point, etc.

A small program having only one segment will have only a mainline segment. Programs created by linking two or more segments together will have a mainline segment followed by the other segments.

Whether or not a segment can be used as a mainline segment is determined by the **typelang** value appearing on the PSECT directive which is discussed in detail in the next section.

THE PSECT DIRECTIVE

SYNTAX:

```
PSECT <name>,<typelang>,<attrev>,<edition>,<stacksize>,
      <entrypoint>,<trapent>
```

LEGAL PSECT STATEMENTS: Any 68000 instruction mnemonic

```
dc
dz
align
vsect
endsect
os9
tcall
ends
```

WARNING: ds can not be used within a PSECT.

PSECT is the program code section. There can only be one PSECT per source file. The PSECT directive initializes all assembler location counters and marks the start of the program segment. All instruction statements and VSECT data reservations must be declared within the PSECT/ENDSECT block.

PSECT may have a parameter list containing a name followed by five or six expressions if the PSECT is to be a "mainline" segment, or it can have no parameter list at all. If a parameter list is provided, the parameter list will be stored in the ROF for later use by the linker to generate the memory module header. If no parameter list is provided, the PSECT name defaults to "program" and all other parameters have default values of zero.

The elements of the PSECT parameter list are as follows:

PARAMETER	DEFINITION
name	Up to 20 bytes for a name to be used by the linker to identify the PSECT. Any printable character may be used except a space or a comma. The name does not need to be unique from other PSECT names, but it is easier to identify PSECTs that the linker has problems with if the names are different.
typelang	A word expression to be used as the executable module type/language word. If the PSECT is not a mainline segment the type/language word must be zero.
attrev	A word expression to be used as the executable module attribute/revision word.
edition	A word expression to be used as the executable module edition word.

PARAMETER	DEFINITION
stacksize	A long word expression that estimates the amount of stack storage required by this PSECT. The linker totals the value in all PSECTs to appear in the executable module and adds the value to any stack storage requirement for the entire program.
entry point	A long word expression to be used as the program entry point offset for PSECT goes here. If the PSECT is not a mainline segment, this should be 0.
trapent	A long word expression indicating the Uninitialized Trap entry point offset. This is used for handling user-mode trap instruction processing. Only give this parameter if the program includes code to handle uninitialized traps. Otherwise, omit this parameter (do NOT use zero!). This parameter is used only in mainline PSECTS.

THE VSECT DIRECTIVE

SYNTAX: `VSECT [REMOTE]`

LEGAL INTERNAL STATEMENTS:
ds
dc
dz
endsect
align

VSECT is the variable storage section which can contain either initialized, uninitialized variable or remotely-addressable variable storage definitions. VSECT directive causes R68 to change the location counter from the code location counter to the data location counters. The data location counter employed depends on the statement used and the presence of the word "remote" after the VSECT directive.

There are three data location counters: one for initialized data, one for initialized data and one for remote data.

The initialized and uninitialized data are intended to be accessed by the 68000 "register indirect with offset" addressing mode: "n(An)". Since the range of this addressing mode is limited to 64k, the total size of these data areas is also limited to 64k.

The remote data area is intended to be addessed with the 68000 "indexed register indirect with offset" addressing mode: "n(An,Xn)". This data area is limited only by the available contiguous system memory. Remote data can not be initialized, but the system does clear the remote area to zeroes when the program starts.

When "ds" is used, the uninitialized data location counter is used. When a remote VSECT is in effect, the "ds" applies to the remote data location counter.

The "dc" and "dz" directives can only be used in a non-remote VSECT. The assembler uses the initialized data location counter for these directives. The constants will appear in the data area of the program when executed. These values can then be modified, if desired.

The "dc" and "dz" directives can also appear outside of a VSECT (in the body of the PSECT). In this case, the constants are assembled into the code area of the program. Do not change constants defined in this manner. To do so would cause the program to be self-modifying and non-re-entrant. Again "dc" and "dz" can not be used in a remote VSECT.

NOTE: The data location counters maintain their values from each VSECT block to the next. Since the linker handles the actual data allocation, there is no facility to adjust the data location counters.

end of chapter 3

ASSEMBLER DIRECTIVE STATEMENTS

ASSEMBLER DIRECTIVE STATEMENTS

Assembler directive statements give the assembler information that affects the assembly process, but usually do not cause code to be generated. Read the descriptions carefully because some directives require labels, labels are optional on others and a few directives can not have labels.

END STATEMENT

SYNTAX: `END`

Indicates the end of a program. Its use is optional since END will be assumed upon an end-of-file condition on the source file. END statements may not have labels.

EQU and SET STATEMENTS

SYNTAX:

```
label EQU <expression>
label SET <expression>
```

These statements are used to assign a value to a symbolic name (the label) and thus require labels. The value assigned to the symbol is the value of the operand, which may be an expression, a name, or a constant. They can be used in any program section.

The difference between the EQU and SET statements is that:

> Symbols defined by EQU statements can be defined only once in the program.
>
> Symbols defined by SET statements can be redefined again by subsequent SET statements.

The EQU statement label name can not have been used previously. The operand can not include a name that has not yet been defined (as yet undefined names whose definitions also use undefined names). Good programming practice dictates that all equates should be at the beginning of the program.

EQU is normally used to define program symbolic constants, especially those used in conjunction with instructions. SET is usually used for symbols used to control the assembler operations, especially conditional assembly and listing control.

EXAMPLES:

```
FIVE    equu    5
OFFSET  equ     address-base
TRUE    equ     $FF
FALSE   equ     0
SUBSET  set     TRUE

        ifne    SUBSET
        use     subset.defs
        else
        use     full.defs
        endc
SUBSET  set     FALSE
```

WARNING: SET can not reference external names. EQU can not reference another EQU that references an external name. Example:

```
joe equ moe
moe equ external
```

FAIL STATEMENT

SYNTAX: `FAIL textstring`

This statement forces an assembler error to be reported. The "textstring" operand is displayed as the error message which is processed in the same manner as R68-generated error messages. Because the entire line following the FAIL keyword is assumed to be the error message, this statement can not have a comment field.

FAIL is most commonly used in conjunction with conditional assembly directives that the programmer sets up to test for various illegal conditions, especially within macro definitions.

EXAMPLE:

```
IFEQ    maxval
        FAIL   maxval cannot be zero
        ENDC
```

IF..ELSE..ENDC STATEMENT

SYNTAX:
```
IFxx  <expression>
   <statements>
[ELSE]
   <statements>
ENDC
```

An important feature of the Macro Assembler is its conditional assembly capability. This is the ability to selectively assemble or not assemble one or more parts of a program depending on a variable or computed value. Thus, a single source file can be used to selectively generate multiple versions of a program.

Conditional compilation uses statements similar to the branching statements found in high level languages such as Pascal and Basic.

The generic IF statement is the basis of this capability. It has as an operand a symbolic name or an expression. A comparison is made with the result: if the result of the comparison is true, statements following the IF statement will be processed. If the result of the comparison is false, the following statements will not be processed until an ENDC (or ELSE) statement is encountered. Hence, the ENDC statement is used to mark the end of a conditionally assembled program section.

Here is an example that uses the IFEQ statement which tests for equality of its operand with zero:

```
IFEQ SWITCH
    .      assembled only if SWITCH = 0
    .
ENDC
```

The ELSE statement allows the IF statement to explicitly select one of two program sections to assemble depending on the truth of the IF statement. Statements following the ELSE statement are processed only if the result of the comparison was false. For example:

```
IFEQ SWITCH
   .    assembled only if SWITCH = 0
   .
ELSE
   .    assembled only if SWITCH is not = 0
   .
ENDC
```

Multiple IF statements may be used, and "nested" within other IF statements if desired. They can not, however, have labels.

IF..ELSE..ENDC STATEMENT (Continued)

There are several kinds of IF statements, each performing a different comparison.They are:

STATEMENT	DESCRIPTION
IFEQ	True if operand equals zero
IFNE	True if operand does not equal zero
IFLT	True if operand is less than zero
IFLE	True if operand is less than or equal to zero
IFGT	True if operand is greater than zero
IFGE	True if operand is greater than or equal to zero
IFP1	True only during first assembler pass (no operand)

The IF statements that test for less than or greater than zero can be used to test the relative value of two symbols if the symbols are subtracted in the operand expression. For example, the following statement will be true if MIN is greater than MAX (the statement literally means IF MAX-MIN < 0):

```
IFLT    MAX-MIN
```

The IFP1 statement causes subsequent statements to be processed during pass 1, but skipped during pass 2. It is useful because it allows program sections which contain only symbolic definitions to be processed only once during the assembly. The first pass is the only pass during which they are actually processed because they do not generate actual object code output. The oskdefs file is an example of a rather large section of such definitions.

For example, the following statement is used at the beginning of many source files.

```
IFP1
use     /d0/defs/oskdefs
ENDC
```

NAM and TTL STATEMENTS

SYNTAX:

```
NAM string
TTL string
```

These statements allow the user to define or redefine a program name and listing title line which will be printed on the first line of each listing page's header. These statements can not have label or comment fields.

The program name is printed on the left side of the second line of each listing page, followed by a dash, then by the title line. The name and title may be changed as often as desired.

EXAMPLES:

```
nam Datac
ttl Data Acquisition System
```

Generates:

```
Datac - Data Acquisition System
```

ASSEMBLER DIRECTIVE STATEMENTS

OPT STATEMENT

SYNTAX: `OPT <option>`

Allows any of several assembler control options to be set or reset. The operand of the OPT statement is one of the characters that represent the various options. An option is denoted by a single character. A preceding "-" turns the specified option off. One exception is the "d" option which must be followed by a number. This statement must not have label or comment fields.

OPTIONS:	[-]c	Prints a listing of conditional assembly lines in an assembler listing. (Default off)
	d <num>	Sets the number of lines per page to <num> (for a listing). (Default 66)
	[-]e	Prints errors. (Default on)
	[-]f	Uses form feed for page eject instead of line feeds. Form feed for top of form. (Default off)
	[-]g	Lists all code bytes generated. (Default off)
	[-]n	Omits line numbers from the assembler listing. This allows more room for comments.
	[-]l	Writes a formatted assembler listing to standard output. If not used, only error messages are printed. (Default off)
	m=<num>	Specifies the microprocessor that the assembler is to be used with: 0 = 68000 - Default 1 = 68010 - Allow 68010 mnemonics/modes 2 = 68020 - Allow 68010 and 68020 mnemonics
	o=<path>	Writes the relocatable output to the specified (mass storage) file.
	[-]q	(quiet mode) Suppresses warnings and nonfatal messages .
	[-]s	Prints the entire symbol table at the end of the assembly listing. (Default off)
	[-]x	Prints macro expansion in assembler listing. (Default off)

PAG and SPC STATEMENTS

SYNTAX:

```
PAG[E]
SPC <expression>
```

These statements are used to improve the readability of program listings. They are not themselves printed, and can not have labels.

The PAG statement causes the assembler to begin a new page of the listing. The alternate form of PAG is PAGE for Motorola compatibility.

The SPC directive puts blank lines in the listing. The number of blank lines to be generated is determined by the value of the operand, which can be an expression, constant or name. If no operand is used a single blank line is generated.

REPT and ENDR STATEMENTS

SYNTAX:

```
REPT <expr>
   <statements>
ENDR
```

This statement can repeat the assembly of a sequence of instructions a specified number of times. The result of the operand expression is used as the repeat count. The expression can not include EXTERNAL or undefined symbols. REPT loops can not be nested.

EXAMPLES:

```
*  20 cycle delay

   REPT 10
   nop
   ENDR

*  Make module size exactly 2048 bytes

   REPT 2048-*-3   compute fill size w/crc space
   dc.b 0
   ENDR
```

DS STATEMENT - DEFINE STORAGE

```
SYNTAX:    <label> ds.s <expr>
                  "s" = "b" (byte)
                        "w" (word)
                        "l" (long).
```

DS is used within VSECTs to declare storage for uninitialized and remote variables in the data area.

When DS is used to declare variables, a label is usually given which is assigned the relative address of the variable. In OS-9, the address is not absolute, so indexed addressing modes are used to access variables. The actual relative address is not actually assigned until the linker processes the ROF. The expression given specifies the size of the variable in bytes, words or longwords (depending of the size given for the DS extension). This value is added to the appropriate unititialized data location counter in order to update it.

The remote data area must be accessed using the "indexed register indirect with offset" addressing mode: "n(An,Xn)".

ASSEMBLER DIRECTIVE STATEMENTS

USE STATEMENT

SYNTAX:
```
USE  pathlist
USE "pathlist"
USE <pathlist>
```

USE statements cause the assembler to temporarily stop reading the current input file. It then requests OS-9 to open another file/device specified by the pathlist, from which input lines are read until an end-of-file occurs. At that point the latest file is closed and the assembler resumes reading the previous file from the statement following the USE statement.

USE statements can be nested (e.g., a file being read due to a USE statement can also perform USE statements) up to the number of simultaneously open files the operating system will allow (usually 29, not including the standard I/O paths).

If an ordinary pathlist (file name) is given, the assembler will search the current working data directory for the file.

If the pathlist is enclosed in quotation marks, the assembler will search for the file in the directory where the source file is located.

If the pathlist is enclosed in angle brackets "< >", the assembler will search for the file in "/dd/defs" on OS-9/68000 systems, or the appropriate directories for cross-compilers.

end of chapter 4

NOTES

PSEUDO-INSTRUCTIONS

PSEUDO-INSTRUCTIONS

"Pseudo-instructions" are special assembler statements that generate object code but do not correspond to actual 68000 machine instructions. Their primary purpose is to create special sequences of code and/or constant data to be included in the program. Labels are optional on pseudo-instructions.

ALIGN STATEMENT

SYNTAX: `ALIGN`

This statement directs the assembler to ensure that the next code generated is aligned in memory on an even word boundary. If the current value of the instruction counter is off word alignment, a zero byte is inserted in the object code. The CPU program counter must always be word aligned for instructions.

This statement is generally used after constant tables or character strings that can have odd lengths are imbedded in the object code.

For more information see the "DC" statement description.

DC STATEMENT

SYNTAX: `<label> DC.S <expression> {, <expression>}`

```
S can be l (long)
         w (word)
         b (byte)
         Default is word
```

The DC (for Define Constant) pseudo-instruction generates sequences of one or more constants (initialized data) of various sizes within the program. The argument(s) is a list of one or more expressions or character strings. If more than one expression or string is given they are separated by commas.

A DC used in a VSECT creates an initialized data variable (read/write) in the process' data area. The initialization value is stored in a special section of the object code and is copied to the appropriate locations in the data area by the FORK system call.

A DC used outside a VSECT is used to create "read-only" constants in the program area that should not be changed by the program.

Character string constants can be any sequence of printable ASCII characters enclosed in double quotes. For the "DC.W" and "DC.L" directives, a string constant is padded with zeroes (on the right end) if it does not fill the final word or long word. Therefore, the "DC.B" form is most natural for strings. It is good practice to use an ALIGN directive after DC.B directives to make sure the instruction counter is left on an even word boundary.

EXAMPLES:

```
dc.b 1,20,"A"

dc.b index/2+1,0,0,1

dc.w 1,10,100,1000,10000

dc.w $F900,$FA00,$FB00,$FC00

dc.b "most programmers are strange people"

dc.b "0123456789"
```

DZ STATEMENT - RESERVE ZERO BYTES

SYNTAX: `<label> DZ.S <expression>`

```
S can be l (long)
         w (word)
         b (byte)
         Default size is word
```

This statement is used to fill memory with a sequence of bytes, each having a value of zero. The 16 bit expression is evaluated and that number of zero values are placed in the appropriate code or initialized data section.

Under OS-9 it is unnecessary and undesirable to reserve zero bytes in the initialized data area (VSECT). The data area is automatically zeroed by the fork system call, so a DZ in the VSECT will only waste space in the object code area.

A DZ used within a PSECT is used to create a "read-only" zero constant that should not be changed by the program. A DZ used within a VSECT is considered as "initialized data" that can be altered by the program.

EXAMPLES:

`dz.b 24`	reserve 24 zero value bytes
`dz.w 1`	reserve 1 zero value word

DO, LO, ORG - DEFINE OFFSET, LINK OFFSET

SYNTAX:

```
<label> DO.S <expression>
<label> LO.S <expression>

             S is l (long)
                  w (word)
                  b (byte)
                  Default is word

ORG <expression>
```

This statement is used to assign an automatically increasing or decreasing set of values to a set of symbolic names. It has nothing to do with memory allocation.

Many times it is desirable to define a group of names with sequentially related values. Some examples are error codes, character sets, stacked variables, etc. DO and LO provide a convenient means of doing this.

Each time a DO is encountered, its label is given the current value of the offset counter. The offset counter is then incremented by the result of the expression multiplied by 1 for a byte, 2 for a word or 4 for a long. When an LO statement is encountered, the offset counter is decremented by the appropriate size and the result is assigned to its label. This is useful in conjunction with the 68000 LINK instruction.

The ORG statement is used to set or change the origin (starting value) of the offset counter.

EXAMPLE:

```
     org $500
 joe do.l 1          is the same as    joe equ 500
 moe do.l 1                            moe equ 504

     org A
A    do.b 1          gives label A the value of its ASCII code
B    do.b 1          gives label B the value of its ASCII code
C    do.b 1          gives label C the value of its ASCII code
D    do.b 1          gives label D the value of its ASCII code
E    do.b 1          gives label E the value of its ASCII code
```

OS9 STATEMENT

SYNTAX: `OS9 <expression>`

This statement is a convenient way to generate OS-9 system calls. It has an operand which is a word value to be used as the request code. The output is equivalent to the instruction sequence:

```
trap #0
dc.w <expression>
```

A file called "sys.l", which is distributed with each copy of OS-9, contains standard definitions of the symbolic names of all the OS-9 service requests. These names are commonly used in conjunction with the OS9 statement to improve the readability, portability and maintainability of assembly language software.

EXAMPLES:

`os9 I$Read`	(call OS-9 "READ" service request)
`os9 F$Exit`	(call OS-9 "EXIT" service request)

TCALL STATEMENT

SYNTAX: `TCALL <vector>,<function>`

This statement is a built-in macro to generate user trap calls. User traps are used to access the OS-9 standard library modules (CIO and math) or user-written trap handlers. TCALL has two arguments, a vector number (zero through 15) and a function code. The output is equivalent to the instruction sequence:

```
trap #<vector>
dc.w <function>
```

For example, the following TCALL is used to access the double-precision floating point comparison in the Math module:

```
tcall T$Math1,T$DCmp
```

NOTE: "tcall #0" is the equivalent to the "OS9" statement explained on the previous page.

end of chapter 5

THE LINKER

THE LINKER

The function of the linker (L68) is to transform the R68 assembler output into a single OS-9 format memory module. A memory module must minimally consist of a module header, a module body and a cyclic redundancy check (CRC).

Many modules require more than this basic information. Program and Trap Handler Modules, for example, require such information as data memory requirement, stack memory requirement, etc. File Manager, Device Driver and Device Descriptor Modules all require special information unique to each. The information required to create each type of memory module is provided to the linker by the contents of the assembly language relocatable output files (ROFs). The linker allows references to occur between modules so one module can reference a symbol in another module. This involves adjustment of the operand parts of many machine-language instructions.

THE ROOT PSECT

A program (or File Manager, Device Driver, etc.) usually consists of many small code segments which, when processed by the linker, form the final executable memory module. Each code segment is called a "psect" The psect is the module unit with which the linker operates. The psect provides the following information to the linker:

1. The identifying information about the psect.
2. The size of the code, data, initialized data and remote memory area.
3. The symbols defined by the individual psect.
4. The symbols referenced by the psect.
5. Relocation information.
6. The actual code and initialized data.

The "root psect" is the psect from which all other references are resolved. The file containing this psect must be named first on the L68 command line. The root psect is distinguished from other psects by the appearance of a non-zero type/language field given in the psect directive of the source file. All other psects processed in the linkage must have zeroes for the type/language field.

The psect directive given for the root psect provides the linker with the name of the psect and prototype module header information. The type/language, attribute/revision, edition number, stack requirement, module entry point offset and uninitialized trap handler entry offset appear in the psect directive. They are used to set up the corresponding entries in the module header. Many of these values can be altered by using linker command line options.

The Subroutine Psect

A zero for the type/language field on the psect directive indicates a subroutine psect. These psects are usually subroutines that provide supporting code for the root psect. Linker library files are simply separately assembled psects, merged together into a single file. Except for the psect name and stack size reservation fields, all fields in the psect directive are zero.

LINKER EXECUTION

When a program is being assembled, the assembler does not know the addresses of names which are external references to other program sections. For example, the "bsr" instruction to a label in another program section can not have its offset computed because the address of the destination label will not be known until all sections are combined by the linker. Therefore, when an external reference is encountered, the assembler will set up information in the ROF which identify the instructions that reference external names. Because the assembler is not aware of what the actual offset within the module will be, each section is assembled as though it starts at offset 0.

The linker takes as its input a number of ROFs produced by the assembler. The linker reads in all of the ROF files and assigns each one relative starting offsets for its data storage space and for its object code space. These addresses remain relative because OS-9 requires programs to be position-independent code with separate position-independent data areas. OS-9 assigns these physical memory areas when a program is loaded and executed.

The linker processes the input files in three phases.

During the first phase, the linker reads all the input files in the order given on the command line. Each psect is checked for validity. The global symbol definitions are entered into the defined symbol table. If a symbol of the same name already exists in the defined symbol table, an error message is generated.

Each global symbol is then checked against the undefined symbol table. If the global symbol defines an external reference, the symbol is removed from the undefined symbol table. Finally, the reference list is examined to identify references from the data area that will need to be adjusted at run time.

After examining the input files, the linker reads the library files. They are processed using the same procedures as used for input files, with minor exceptions. A psect appearing in a library file is retained for the final module only if the psect defines an as yet undefined symbol. After each psect in a library file is processed, the undefined symbol table is examined to see if any symbols are still undefined. If so, library file processing continues. If not, the next psect in the library file, or the next library file (whichever is appropriate) is processed.

A **symbol psect** is handled as a special case by the linker. It contains no code or data, only symbols defining constants. When a symbol psect is processed, only the symbols defining as yet undefined symbols are placed in the defined symbol table. Symbol psects are used in the "sys.l" system library file to define system constants and offset values. This procedure minimizes the amount of symbol table memory required for linking modules against the system library.

The second phase is the linker allocation phase. The size of the code, data, initialized data and remote memory areas are determined. The offsets of all code symbols are assigned based on each psect's position in the final module.

If the "-a" option is turned on, a symbol reference list is examined to determine if any code falls outside of the 16-bit offset addressing mode limit. If any portion of the psect is farther than 32k bytes from the destination, an entry is reserved in the jumptable.

The offsets of all data symbols are assigned. The uninitialized data memory is assigned first, followed by the initialized data memory (this includes the linker-generated jumptable). Finally the remote memory area is assigned. The total size of the uninitialized and initialized data areas can not exceed 64k bytes. The size of the remote memory area is limited only by the amount of contiguous free memory in the system.

The linker creates the output module during the third phase. The module header for the appropriate module type is created and written to the output file. Each input psect is re-read from the appropriate input or library file. The code and initialized data segments are read into an internal buffer.

The reference list in the psect is read to determine the locations of all operands referencing external symbols. These operands are then adjusted to reflect the destination's position in the output module.

If the "-a" option is in effect and the reference can not reach the destination with the 16-bit offset addressing mode, the reference is changed to reference a jumptable entry.

The code and initialized data segments are then written to the output file. As each segment is written out, the OS-9 module CRC is calculated. The CRC is then written into the output module when all psects have been processed.

LINKER LIBRARY FILES

Library files are created by concatenating one or more ROFs into a single file. To change a single psect in a library file, the entire library must be re-created from the ROFs, substituting the new psect for the old.

The linker performs only one pass over the input files to locate symbol definitions. Because of this, the order in which the psects appear in the library file is very important. The psects must be ordered so that the references are generally forward references. Consider the following example:

```
psect main_c
     defines:          main
     references:       sub_1

psect sub1_c
     defines:          sub_1, sub1a
     references:       sub2

psect sub2_c
     defines:          sub2
     references:       printf
```

The psect "sub1__c" must appear in the library before any psect containing a symbol that "sub1__c" references. If the "sub2__c" psect were to appear before the "sub1__c" psect, the symbol definition for sub2 would not be found. Remember, a psect appearing in a library file is retained for the final module only if the psect defines an as yet undefined symbol.

LINKER DEFINED AND LINKER RECOGNIZED SYMBOLS

The linker will define some symbols at link time that can not be determined until the final code and data offsets are determined. The linker-defined symbols are:

SYMBOL	INDEX REGISTER	REFERENCES
`_jmptbl`	a6	Offset to jumptable.
`end`	a6	Last data offset assigned.
`bname`	pcr	Offset from the beginning of the module to module name.
`btext`	pcr	Offset from pc to beginning of the module.

The linker recognizes certain global labels as overrides to selected fields in the module header. The linker will place the value of these symbols into the appropriate field of the module header:

`_sysedit`	Edition number to place at M$Edit.
`_sysperm`	Permision value to place at M$Accs

These symbols are typically set with the equ directive as:

```
_sysedit: equ 21              ;edition number

_sysperm: equ PRead_|Read     ;module acess permissions
```

THE LINKER COMMAND LINE

The linker is called using the command line:

```
L68 [options] <mainline> [<rof2> [<rofN>] ] [options]
```

The <mainline> argument is the pathlist of the mainline segment (see Chapter 3) from which external references are resolved and a module header is generated. A mainline module is indicated by setting the type/lang value in the PSECT directive to a non-zero value.

Names of additional ROFs (ROF2 through ROFn) to be used in the linkage process follow the mainline ROF pathlist. No other ROF can contain a mainline PSECT. The mainline and all subroutine files will appear in the final linked object module whether actually referenced or not. There is no limit to the number of ROFs and a limit of 32 library file specification per linkage. All L68 input files must be in relocatable object format (ROF).

PSECTS that contain no data or code are handled by L68 in a special way. This type of PSECT contains only symbols that define constants (i.e., equ). Only the symbols that define external references are placed in the linker's symbol table. If used, this type of PSECT is the last PSECT given and handles remaining unresolved references. An example of this is the "sys.l" file (system definition library).

LINKER COMMAND LINE OPTIONS

The following options can appear on the command line:

OPTION	DESCRIPTION
-a	Converts out-of-range BSRs and pc-relative LEAs to jumptable references. BSRs that address labels over 32k bytes distant will automatically be converted to JSRs using a jump table (in the initialized data area) that contains the desired destination address. LEAs will be changed to move instructions that move the destination from a jump instruction in the jump table. The linker automatically builds the required jump tables and includes them in the output file. This allows large programs to overcome the +/- 32K byte offset limit of BSR instructions without violating the OS-9 requirement for position independent code.
-e=<n>	Sets the module edition number. <n> is used for the edition number in the final output module. 1 is used if this option is not given.
-g	Outputs symbol module for use by the user debugger. The symbol module file name is the output file name with ".stb" appended. If a directory named "STB" is present in the current execution directory, the symbol file is placed there. Otherwise, it is placed directly in the current execution directory.
-j	Prints jump table calculation map (see description in the -a option section).

OPTION	DESCRIPTION
-l=<path>	Uses <path> as a library file. A library file consists of one or more merged assembly ROF files. Each PSECT in the file is checked to see if it resolves any unresolved references. If so, the module is included in the final output module, otherwise it is skipped. No mainline PSECTs are allowed in a library file. This option can be repeated up to 32 times in one command line to specify multiple library files. Library files are searched in the order given on the command line. The standard definition files are "sys.l" for assembly language or "clib.l" for the C compiler.
-M=<size>	Adds <size> kbytes to the stack memory allocation ("k" is optional, i.e., "-M=4" and "-M=4k" are equivalent).
-m	Prints linkage map indicating base addresses of the PSECTs in the final object module.
-n=<name>	Uses <name> as module name.
-o=<path>	Writes linker object (memory module) output to the file specified. The last element in <path> is used as the module name unless overridden by the "-n" option.
-p=<n>	Sets the permission word in the module header to <n>. <n> must be hexadecimal.
-r	Outputs a raw binary file for non-OS-9 target system. The output will not be in memory module format.
-r=<n>	Outputs a raw binary file for non-OS-9 target systems with an object code base address at absolute address <n>, which must be a hexadecimal address. The base address is used to make absolute addressing references operate correctly.
-s	Prints a list of relative addresses assigned to symbols in the final object module. Output is in numeric. This option is usually used with the "-m" option.
-w	When used with -s will display symbols in alphabetic instead of numeric order (see above).
-z	Reads module names from standard input.
-z=<file>	Reads module names from <file>.

LINKING CODE FOR NON-OS-9 SYSTEMS

The linker can generate "raw" code to run in non-OS-9 environments. The output is a pure binary file which is not in OS-9 memory module format.

The linker "-r" option is used to create raw output files. The "-r=n" option is used where "n" is a hex address to place modules in ROM. The address is used to make absolute references come out correctly.

Since it is assumed that the code will not be executed via the OS-9 "FORK" system call (which performs data area initialization), no initialized data may be used ("dc" in a VSECT).

The user's initialization code must set up the stack pointer (A7) to point to a stack RAM area, and the A6 register must point to the beginning of a global/static RAM area (VSECT) which should be initialized to zeros.

end of chapter 6

OS-9 PROGRAMMING TECHNIQUES

One of OS-9's main features is its powerful memory management capabilities using software or software/hardware methods. This results in much more efficient and flexible use of system memory than in other operating systems.

In order for these techniques to work properly, the programmer is required to follow certain rules when writing assembly language programs. Programmers who use only high-level languages such as C, Basic, Pascal, etc., need not be as concerned with these rules because the compilers automatically carry them out.

The key to being able to effectively use these methods is to have a good knowledge of the environment OS-9 provides for programs and also the instructions and addressing modes of the 68000 processor.

RULE 1: **All executable code must be in memory module format.**

The OS-9 memory module is the basis of both memory management and the advanced modular programming techniques the operating system supports. The assembler and linker automatically generate the module header and CRC check values. For detailed information concerning memory modules, see the "OS-9/68000 OPERATING SYSTEM TECHNICAL MANUAL".

RULE 2: **Program and data areas must be separate.**

All object code for a program is located in a memory module which is "read-only". Programs should never modify themselves (this is good programming practice anyway). Therefore, a separate memory area is used for variables. Every process has a unique data area. Every process does not necessarily have a unique program memory module. This allows two or more tasks to "share" the same copy of a program if they are running the same program. This technique is an automatic function of OS-9 that usually results in much more efficient use of available memory.

RULE 3: **All object code must be position-independent.**

OS-9 must be able to dynamically map a memory module to any block of physical addresses to allow a process to access more than one module at the same time, and also allows memory management on systems that don't have memory management hardware with address relocation functions.

Writing position-independent code involves using only PC-relative addressing in branches and accessing constant tables. Absolute memory addresses should never be used in a program.

RULE 4: **All data storage must be position-independent.**

The address of the program's data area is assigned by OS-9 at the time the process is started by the FORK system call. Use of a position-independent data area lets OS-9 run on systems with limited or nonexistent memory management hardware. As with the object code module, absolute addressing of variables is not permitted. Instead, OS-9 programs use the convention that register A6 is a pointer (base address register) to the program's data area, and all addressing of variables uses the 68000 indexed addressing modes. The initialized and uninitialized data area are accessed by the "register indirect with offset" addressing mode: "n(An)". The remote data area is accessed by the "indexed register indirect with offset" addressing mode: "n(An,Xn)".

PROGRAM AND DATA MEMORY REFERENCES

OS-9 does not limit the memory size of a program's code or data. However, due to the characteristics of the 68000 architecture, certain restrictions exist. These depend on the addressing modes used. Because of the software techniques OS-9 uses to provide a multi-user and multi-tasking environment, user state programs can not use either the "absolute short" or "absolute long" addressing modes for addressing code and data memory.

All references to the program code must use a "pc-relative" addressing mode:

`n(pcr)`	Relative with Offset
`n(pcr,Xn)`	Relative with Index and Offset or any of the relative branch instructions

All references to the program data must use a "register indirect" addressing mode:

`(An)`	Register Indirect
`(An)+`	Postincrement Register Indirect
`-(An)`	Predecrement Register Indirect
`n(An)`	Register Indirect with Offset
`n(An,Xn)`	Indexed Register Indirect with Offset

The offsets of these addressing modes are 16-bit signed offsets. This limits the usefulness of the offset to +/-32k bytes. Many non-trivial programs rapidly exceed this 32k limit. Because it is undesirable to require assembly language programmers and compilers to generate worst-case code all the time, the OS-9 assembler and linker provide facilities to access distant program and data addresses.

OS-9 PROGRAMMING TECHNIQUES

DATA AREA REFERENCES

The address of the data memory for a process is placed in the A6 register by OS-9. The labels appearing in the program's VSECTs represent offsets. When applied against the A6 register, these labels yield the address of the desired data. Since the offset is limited by the hardware addressing mode to a 16-bit signed value, the offset can address only 32k bytes. To fully utilize the 64k byte range that an unsigned offset would provide, the linker automatically starts assigning the data storage offsets from $8000. When OS-9 assigns data memory for a process, the A6 register is automatically adjusted to point 32k bytes PAST the actual base of the data memory. This method allows a full 64k bytes of addressability from the "register indirect with offset" addressing mode.

The 32k data memory base adjustment is done only for user state program and trap handler modules. No adjustment is made for system state modules such as device drivers, file managers, etc.

Depending on the storage reservation directive, the assembler assigns the value of the appropriate location counter to labels appearing in a non-remote VSECT. A label on a "DS" directive references the uninitialized data area. A label on a "DC" or "DZ" directive references the initialized data area.

The total size of the initialized and uninitialized data memory can not exceed 64k bytes. To do so would cause the memory beyond 64k bytes from the base pointer to be unaddressable with the "register indirect with offset" addressing mode.

The following is an example of referencing the initialized and uninitialized data areas:

```
        vsect
varname ds.l 1   ;an integer variable
counter dc.l 500 ;an integer variable initialized to 500
        ends
        .
        .
        move.l varname(a5),d0 ;access the uninitialized data memory
        cmp.l  d0,counter(a6) ;access the initialized data memory
```

The only way to offset beyond the 64k data memory limit is to use the "indexed register with offset" addressing mode. The offset for this addressing mode is only 8-bits signed, which renders it useless for this purpose. The index register, however, can be loaded with a 32-bit constant representing the offset to the data. This method yields a 4.2 gigabyte unsigned offset.

When processing a remote vsect, the assembler assigns labels the value of the remote location counter. The linker will then assume the label to be a 32-bit offset into the data memory area. Any references to the label are then used as long offsets to be applied to the A6 register to access the data:

```
         vsect remote
bigone:  ds.l 100000              ;declare a very large array
         ends
         .
         .
         move.l #bigone,d0        ;get offset into bigone
         add.l  d4,d0             ;add subscript
         move.l 0(a6,d0.l),d2     ;get the array value
```

The address of "bigone" can be determined by:

```
move.l a6,a0         ;get the base address of the data
adda.l #bigone,a0    ;add in the offset to bigone
move.l (a0),d2       ;get the array value
```

CODE AREA REFERENCES

Many of OS-9's extraordinary capabilities are due to the memory module concept. All OS-9 object code must be in memory module format. Use of memory modules is simple because the L68 linker automatically generates them.

Another important requirement for OS-9 object code is position-independence. Use of position-independent code (PIC) is essential. It allows OS-9 to dynamically add or remove modules from a process' memory space. PIC allows OS-9 to load a program at any address where free memory is available. Absolute addressing should never be used.

Fortunately, the 68000 instruction set is generally well suited for writing PIC. PIC programming techniques require that only program counter relative (PCR) addressing modes be used to access the program object code area. The BSR, Bcc, and DBcc instructions do this inherently. Notice that the 68000 addressing scheme does not allow an operand addressed as PCR to be modified. This enforces the OS-9 design philosophy that no program should modify itself.

When addressing constants, constant tables or addresses of routines within the program object code, the PCR addressing modes ("d(PC)" and "d(PC,Xi)") can only be used on instructions that DO NOT alter their objects. Directly stated, PCR addressing modes can never appear as a destination adress or as an address of data to modify. The 68000 addressing modes themselves discourage self-modifying programs.

The LEA and PEA instructions can be used with PCR addressing modes to obtain actual addresses within the program area at run time. For example, to obtain the address of a constant table the following instruction can be used:

```
lea table (PC),a2
```

The offset in the PCR addressing mode is limited to a 16-bit signed value. This limitation restricts the use of this addressing mode to destinations within 32k bytes of the reference. Because many non-trivial programs easily exceed this limit, the linker provides a facility to overcome this limitation.

The "-a" option of the L68 linker causes the linker to direct certain PCR references to a jumptable in the data area. For each BSR instruction that does not reach its destination address, the linker will replace the BSR instruction with a JSR instruction. The destination of this instruction references a jump table.

The linker creates the jumptable as initialized data, the size of which is added to the total initialized data allocation for the program. Each jumptable entry is an absolute long JMP instruction whose destination is initially an offset from the beginning of the program module to the reference.

The relocation information placed in the module by the linker is then used by the kernel to adjust the offsets in the jumptable to reflect the absolute address of the actual reference. Only long (16-bit offset) BSR instructions are affected. All other branch style instructions (Bcc, DBcc, etc.) are still limited to the 32k-byte restriction.

The following is an example of a modified BSR instruction. Consider the jumptable fragment:

```
jmptbl+0    jmp printf
jmptbl+6    jmp fprintf
jmptbl+12   jmp sprintf
```

An out of range bsr:

```
bsr fprintf
```

would be replaced by:

```
jsr _jmptbl+6(a6)
```

The "-a" option also causes LEA instructions involving a PCR reference to be directed through the jumptable. The LEA instruction is commonly used by OS-9 programs to determine the address of a table, or in the case of a C program, a function. Often this instruction can not address its destination because of the range limit. In this case, each LEA instruction that does not reach its destination address is replaced by a move which references the destination address appearing in the jumptable.

For example, an out of range LEA:

```
lea fprintf(pcr),a0
```

would be replaced by:

```
move.l _jmptbl+6(a6),a0
```

Only one jumptable entry is created for each unique unreachable reference, regardless of the number of times a reference could not be reached. This allows resolution of the reference without changing the size of the code. Both the original code and the substitute code are four bytes long.

end of chapter 7

EXAMPLE PROGRAM

The following example is the assembly language program UpDn: a program that converts the case of input to either upper or lower. This program is provided to give an example of what an assembly language program should be in the way of structure and form.

```
Microware OS-K RASM V0.2  84/6/14  9:20    updn.a            page  1
UpDn - OS-9/68000 Example Assembly Language Program

                    nam     UpDn
                    ttl     OS-9/68000  Example Assembly Prog
*********************************
* This program converts characters from upper case to lower case
* (default) or from lower case to upper case (with -u option)

* Intended as a comparative example to the 6809 version found
* in the "OS-9/6809 Editor/Assembler/Debugger User's Manual

                    use     defsfile

 00000001 Edition   equ     1
 00000101 Typ_Lang  equ     (Prgrm<<8)+Objct
 00008000 Attr_Rev  equ     (ReEnt<<8)+0
                    psect   updn,Typ_Lang,Attr_Rev,Edition,0,UpDn

 00000000 StdIn     equ     0              standard input path
 00000001 StdOut    equ     1              standard output path
 00000002 StdErr    equ     2              standard error path

                    vsect
 00000000 Char      ds.b    1              one character I/O buffer
0000    41 LowBound dc.b    'A'            low bound to convert
0001    5a HiBound  dc.b    'Z'            upper bound to convert
 00000002           ends

0000=5379 HelpStr   dc.b    "Syntax: updn [-u]",C$LF
0012=4675           dc.b    "Function: converts upper to lower ",C$LF
0043=4f70           dc.b    "Options:",C$LF
004c=2020           dc.b    " -u : converts lower to upper",C$LF,C$CR
 00000071 HelpLen   equ     *-HelpStr

*  Entry Point and Initialization

0072 ddfc UpDn      adda.l  #$8000,a6  adjust stack <<<Remove this>>>
0078 600e           bra.s   PrsOpt10       parse options

007a 101d PrsOpt    move.b  (a5)+,d0       get parameter byte
007c b03c           cmp.b   #'-',d0        parameter leadin?
0080 671e           beq.s   PrsOpt20       branch if so
0082=b03c           cmp.b   #C$CR,d0       carriage return?
0086 6606           bne.s   PrtHelp        abort if not
0088 51cd PrsOpt10  dbra    d5,PrsOpt      until no more option bytes
008c 602a           bra.s   UpDn10
```

```
Microware OS-K RASM V0.2  84/6/14  9:20     updn.a          page  2
UpDn - OS-9/68000 Example Assembly Language Program

008e 7002 PrtHelp    moveq    #StdErr,d0      standard error path
0090 41fa            lea      HelpStr(pc),a0  help message string
0094 223c            move.l   #HelpLen,d1     length of string
009a=4e40            os9      I$WritLn        output help message
009e 604c            bra.s    UpDn90          exit

00a0 5345 PrsOpt20   subq.w   #1,d5           decr option cnt
00a2 65ea            bcs.s    PrtHelp         abort if end of parameters
00a4 101d            move.b   (a5)+,d0        get option character
00a6 0a00            eori.b   #'u',d0         is it a "-u" option?
00aa 0200            andi.b   #^('a'-'A'),d0  (ignore case difference)
00ae 66de            bne.s    PrtHelp         abort if not
00b0 3d7c            move.w   #'az',LowBound(a6) reset convert bounds
00b6 60d0            bra.s    PrsOpt10        check for other options

00b8 7000 UpDn10     moveq    #StdIn,d0       from standard input
00ba 7201            moveq    #1,d1           read one character
00bc 41ee            lea      Char(a6),a0     into "Char"
00c0=4e40            os9      I$Read
00c4 6520            bcs.s    UpDn80          abort if error
00c6 102e            move.b   Char(a6),d0     get character
00ca b02e            cmp.b    LowBound(a6),d0 in range?
00ce 650c            blo.s    UpDn20          branch if not
00d0 b02e            cmp.b    HiBound(a6),d0  in range?
00d4 6206            bhi.s    UpDn20          branch if not
00d6 0a2e            eori.b   #'a'-'A',Char(a6) convert characters case
00dc 7001 UpDn20     moveq    #StdOut,d0      to standard output
00de 7201            moveq    #1,d1           write the character
00e0=4e40            os9      I$WritLn
00e4 64d2            bcc.s    UpDn10          repeat if no error

00e6=b27c UpDn80     cmp.w    #E$EOF,d1       end of file error?
00ea 6602            bne.s    UpDn99          abort if not
00ec 7200 UpDn90     moveq    #0,d1           return without error
00ee=4e40 UpDn99     os9      F$Exit          exit

 000000f2            ends

Errors: 00000
Memory used: 11k
Elapsed time: 11 second(s)
```

end of appendix a

ERROR CODES

When RMA detects an error, it prints an error message in the listing just before the source line containing the error. It is possible for a statement to have two or more errors, in which case each error is reported on a different line preceding the erroneous source line.

If the assembler listing is inhibited by the absence of the "-l" option, error messages and printing of erroneous lines still occurs. At the end of the assembly, the total number of errors and warnings are given as part of the assembly summary statistics. The error messages, erroneous source lines, and the assembly summary are all written to the assembler task's error/status path which may be redirected by the shell. For example:

```
rma sourcefile  o=sourcefile.o > src.error
```

Note that calling the assembler with the listing and object code generation both disabled by the absence of the " -l -o " options can be used to perform a quick assembly just to check for errors. This allows many errors to be found and corrected before printing of a lengthy listing. For example:

```
rma sourcefile
```

Sometimes the assembler will stop processing of an erroneous line so additional errors following on the same line may not be detected, so corrections should be made carefully.

Error messages consist of brief phrases which describe the kind of error the assembler detected. Each error message is explained in detail in this appendix.

ERROR MESSAGE	DESCRIPTION
`Bad label`	The statement's label has an illegal character or does not begin with a letter.
`Bad Mnemonic`	A mnemonic was found in mnemonic field that was not recognized or was not allowed in the current program section.
`Bad number`	The numeric constant definition contains a character that is not allowed in the current radix.
`Bad operand`	An operand expression is missing or incorrectly formed.

ERROR MESSAGE	DESCRIPTION
`Bad operator`	An arithmetic expression is incorrectly formed.
`Bad option`	An option is unrecognized or incorrectly specified.
`Bracket missing`	The opening or closing bracket is missing.
`Can't open file`	A problem was encountered opening an input file.
`Can't open macro work file`	A problem was encountered opening a macro work file.
`Comma expected`	A comma was expected but not found.
`Conditional nesting error`	A mismatched if/else/endc conditional assembly directive was found.
`Constant definition`	A constant definition is incorrectly formed.
`ENDM without MACRO`	An ENDM was found, with no matching MACRO.

ERROR CODES

ERROR MESSAGE	DESCRIPTION
`ENDR without REPT`	An ENDR was found, with no matching REPT.
`Fail <message>`	A fail directive was encountered.
`File close error`	A problem was encountered closing an input file.
`Illegal addressing mode`	The addressing mode can not be used in the instruction.
`Illegal external reference`	External names can not be used with assembler directives. If an operand expression contains an external name, the only operation allowed in the expression is binary plus and minus.
`Illegal index register`	The register can not be used as an index register.
`Label missing`	This statement is missing the required label.
`Macro arg too long`	The Macro argument is too long. No more than 60 characters total can be passed to a macro.
`Macro file error`	A problem was encountered accessing the macro work file.

ERROR MESSAGE	DESCRIPTION
`Macro nesting too deep`	The Macro calls are nested too deeply. Macro calls may only be nested up to 8 levels deep.
`Nested MACRO definitions`	A macro can not be defined inside a macro definition.
`Nested REPT`	Repeat blocks can not be nested.
`New symbol in pass two`	See symbol lost.
`No input files`	An input file must be specified.
`No param for arg`	A macro expansion is attempting to access an argument that was not passed by the macro call.
`Phasing error`	A label has a different value during pass two than it did during pass one.
`Redefined name`	The name appears more than once in the label field other than on a SET directive.

ERROR CODES

ERROR MESSAGE	DESCRIPTION
`Symbol lost?`	Assembler symbol lookup error. The error could be caused by symbol table overflow or bad memory.
`Too many args`	Too many arguments were passed to the Macro. No more than 9 arguments may be passed to a macro.
`Too many object files`	Only one "-o=" command line option is allowed.
`Undefined org`	"*" (program counter org) can not be accessed within a VSECT.
`Unmatched quotes`	A beginning or ending quotation mark was expected but not found.

LINKER ERROR MESSAGES

The following is a comprehensive list of the error, warning and informational messages issued by the OS-9/68000 linker (l68). In this section the following syntax conventions are used:

`'<file>'`	represents the actual file name in question.
`'<n>'`	represents the actual number in question.
`'<char>'`	represents the actual character in question.

ERROR MESSAGE	DESCRIPTION
`'<file>' contains a 6809 module`	A module from the 6809 assembler was encountered.
`'<file>' contains assembly errors`	A module was encountered that had assembly errors. Fix the errors and re-link.
`'<file>' contains no root psect`	The first file given on the command line must contain a root psect. A root psect is the psect from which all references are resolved. A root psect is specified by non-zero type and language fields in the module's psect directive.
`'<file>' created by assembler too new for this linker`	The r68 and l68 programs are not compatible editions. Be sure the correct programs are installed in the execution directory.
`'<file>' is not a relocatable module`	The relocatable module header in '<file>' was either not present or incorrectly formed. All relocatable object headers start with the bytes: $DE $AD $FA $CE. Use the "dump" utility on the input file to verify this. The most likely cause of this error is the wrong file was given on the command line.

ERROR CODES

ERROR MESSAGE	DESCRIPTION
`'<file>' rof<4 and code>32k. Must be re-assembled.`	This message is caused when the linker processes an old version of assembler output that contains more than 32k of code. Re-assembly of the source file will fix the problem.
`bad syscrc size`	This is an internal linker error. Contact Microware if this error can be reproduced at will.
`can't create output file`	The output file for the module (given by the -o= option) can not be created. Possible causes are no access permissions or no disk space.
`can't create symbol file`	The symbol file for the module can not be created. Possible causes are no access permissions or no disk space.
`can't open '<file>'`	One of the input files given could not be opened. Possible causes are no access permissions, non-existent file or no free memory.
`can't open '<file>' name file`	The -z=<file> could not be opened. Possible causes are no access permissions, non-existent file or no free memory.
`can't reopen input file '<file>'`	This is an internal linker error. Contact Microware if this error can be reproduced at will.

ERROR MESSAGE **DESCRIPTION**

`duplicate symbol names`

The linker has determined that the same symbol name appears in more than one psect in the allocation of the final module. Consider the following program fragment:

```
main()
{
        strcat();
}
strlen()
{
        return;
}
```

When compiled and linked, linkage will fail and return the messages:

```
Symbol 'strlen' defined by psect 'strings_c' in file '/dd/lib/clib.l' has
     already appeared in psect 'prog_c' in file 'ctmp.001543.r
Symbol 'strcat' from psect 'strings_c' in file '/dd/lib/clibn.l caused name
     clashes.
```

Since both 'prog__c' and 'strings__c' define the symbol 'strlen', the linker can not resolve the reference to 'strcat' without causing the definition of 'strlen' to be ambiguous. The first message indicates that 'strlen' was defined in both 'prog__c' and 'string__c'. The second message identifies the symbol that the linker was attempting to resolve when it found the name clash.

`error reading input file '<file>'`

The linker could not read the input file. Either a physical error occurred or the input file was incorrectly formatted. All input files must be output from the assembler.

`error writing file '<file>'`

The linker could not write the output file. Possible causes are disk errors or media full.

ERROR CODES

ERROR MESSAGE	DESCRIPTION
`initialized data (or jumptable) allowed only on program or trap handler modules`	Initialized data is supported only for program modules (entered by F$Fork) or trap handler modules (entered by F$TLink). Modules such as system modules, device drivers and file managers can not have initialized data. Initialized data is generated by the C Compiler when C initializers appear for static or global data. Initialized data is generated by the assembler when a data initialization directive (dc, dz, etc.) appears in a vsect.
`jmp total > guess (<n>/<n>)`	This is an internal linker error. Contact Microware if this error can be reproduced at will.
`no data storage allocation (vsect) allowed on non-object modules`	Only modules of type "object code" can contain data storage allocation. Other module types (usually language runtime interpreters) define data storage allocations in a different manner.
`no initialized static data allowed on raw output`	Initialized data can not be allocated to a program designed to run without OS-9. Uninitialized data is allowed. The linker prints the size of the uninitialized data requirement for raw modules.
`no root psect found`	The first module given on the command line must contain a root psect. A root psect is a module in which the psect directive indicates a non-zero type and language word. A zero type and language word means a subroutine module. The root psect is the psect from which all references are resolved.
`non-remote data allocation value exceeds 64k`	See the discussion in the manual on "data area references" for a description of the linker memory limits.

ERROR MESSAGE	DESCRIPTION
`odd count for crc`	This is an internal linker error. Contact Microware if this error can be reproduced at will.
`operand size error`	This message occurs when an operand value exceeds the legal range for the size of the operand. It is displayed with additional text such as:

```
168: error - operand size error
The value of symbol 'funny' ($193) is too large for a byte operand.
The offending operand is at offset $13c1 in the code area of psect 'main'in
     file 'pt.r'.
```

In this case, the value for the symbol "funny" is (in hex) $193. This value is too large to fit in a byte-size operand. The next line indicates where in the psect the operand appears. In this case the operand appears at offset (in hex) $13c1 in the code area. The location counter field on the assembly listing is the offset value. Finally, the offending psect name and the file in which the psect appears is displayed.

`out of memory`	The linker can not obtain enough memory to do the linkage. Memory usage requirements depend on many factors: number of input files, number of psects, number of global symbols and undefined references. The largest use of memory is during the second pass when each psect's references must be adjusted for the final program module. To do this step, the linker must be able to get as much memory as the largest psect used. A psect that is 128k bytes long requires a 128k buffer to link.
`reference location error (<n>)`	This is an internal linker error. This is caused by information from the assembler that the linker does not expect. If this happens, be sure the assembler and linker are properly installed on the system from the original distribution medium. Contact Microware if this error can be reproduced at will.

ERROR CODES

ERROR MESSAGE	DESCRIPTION
`root psect found in both <file1> and <file2>`	Only one root psect is allowed for a program. A root psect is defined as a psect in which the Type/Language field is non-zero. The root psect is the initial psect from which all external references are resolved.
`symbol 'name' not found during pass 2`	This is an internal linker error. Contact Microware if this error can be reproduced at will.
`too many data references <n>`	This is an internal linker error. Contact Microware if this error can be reproduced at will.
`unknown option -<char>`	An option given is not an option recognized by the linker.
`unknown reference type <n>`	This is an internal linker error. Contact Microware if this error can be reproduced at will.

`unresolved references`	The symbols previously listed by the linker are not defined by a psect given on the command lines or in libraries. This is commonly caused by improperly ordered library files. See Chapter 6 and the discussion of Linker Library Files for details on library searching. Additional error messages such as the following normally appear previous to this message:
	`Symbol 'funny' unresolved.`
	`Referenced [<n> times] by psect 'main' in file 'pt.r'`

end of appendix b

NOTES

ROF FILE FORMAT

RELOCATABLE OBJECT FILE FORMAT

The object code output by the assembler must be processed by the linker before the code is executable. The assembler writes the object code in a special relocatable object file format (ROF) to allow the linker to link together separately assembled modules into a single executable module. The ROF contains information such as the global data definitions, code entry points, external references, actual object code, and initialized data.

It is unlikely that a programmer would have to deal with the internals of an ROF. The information given here is for informational purposes. The "RDUMP" program can be used to extract this data from existing relocatable files.

HEADER SECTION

ROF Sync bytes (4 bytes)

Sync bytes used by L68 to recognize an ROF.

Type/Language (2 bytes)

The type/language word from the PSECT. This word is used by the linker to determine the desired OS-9 module format. If the word is zero, the routine is assumed to be a subroutine type module. Only the mainline segment can have this word be nonzero.

Attribute/revision (2 bytes)

Attribute/revision word to place in the OS-9 module. This word is only meaningful on a mainline segment.

Assembly valid (2 bytes)

This will be nonzero if assembly errors occurred. This word is used to prevent the linker from linking erroneous modules.

Series (2 bytes)

This is used to indicate to the linker which version of the assembler is to be used. This word is used to prevent problems that could occur in mixing different versions of the linker and assembler.

Date/Time Assembled (6 bytes)

This indicates the date and time of assembly.

Edition number (2 bytes)

This is the OS-9 edition number to be placed in the output module for mainline segments. For non-mainline segments this word is informational only.

Size of static storage (4 bytes)

This tells the linker how much static data storage to reserve for the module. This value is determined from the total size of the "ds" directives in the VSECTs.

Size of initialized data (4 bytes)

This tells the linker how much initialized data the module contains. The size is determined by the total size of all the "ds" directives in the VSECTs.

Size of the object code (4 bytes)

This is determined from the size of code assembled.

Size of stack required (4 bytes)

This tells the linker how much stack space is required by the module. The value is obtained directly from the PSECT directive.

Offset to entry point (4 bytes)

This value is the offset to the entry point in the object code. The offset is relative to the beginning of the module. The value is obtained directly from the PSECT directive.

Offset to uninitialized trap entry point (4 bytes).

Name of module (variable length)

This is a null terminated ASCII string taken directly from the PSECT directive. The name is used by the linker to identify the PSECT in case of an unresolved reference or other error.

External definition count (2 bytes)

The count indicates the number of external references that will follow. The external definitions are placed in the linker's symbol table and can be referenced by any other module.

EXTERNAL DEFINITION SECTION

Each external definition has the following format:

1-9 bytes	Name - null terminated
2 bytes	Type/location flag
4 bytes	Symbol value

Type/location definitions:

0 - references to symbol reference static data area
1 - references to symbol reference initialized data
4 - references to symbol references code area
6 - references to symbol reference an absolute (equ)

Object code: object code for the module.

The size of this section is indicated by the "Size of Object Code" bytes defined above.

Initialized data (variable length)

The size of this section is found in the "Size of Initialized Data" defined above.

NOTE: Either or both the object code section or the initialized data section may be omitted. The sum of both can not exceed 64k per PSECT. If both are missing and the static data count is zero, the module can only contain absolute (equ) symbols. In this case, the linker will extract only those symbols that resolve an external reference. The OS-9 "sys.l" library module is an example. It contains nothing but equ'ed symbol definitions.

Number of external references (2 bytes).

EXTERNAL REFERENCE SECTION

1-9 bytes	Null terminated name string
2 bytes	Reference count - Indicates the number of references following.
2 bytes	Type/location flag
4 bytes	Reference offset - offset into code or initialized data section where the reference appears.
2 bytes	Number of local references - these are references in the code or initialized data areas that reference code or initialized data in the PSECT.

LOCAL REFERENCE SECTION

2 bytes	Type/Location flag
4 bytes	Local Offset

end of appendix c

INDEX

A

Address counters 3-5
Address mode syntax 1-10
ALIGN statement 5-1
Assembler Directive Statements
  definition 4-1
  DS statement 4-10
  END statement 4-1
  EQU statement 1-5,4-2
    function 4-2
    syntax 4-2
  FAIL statement 4-3
  IF/ELSE/ENDC statement 4-4
    IFEQ 4-5
    IFGE 4-5
    IFGT 4-5
    IFLE 4-5
    IFLT 4-5
    IFNE 4-5
    IFP1 4-5
  NAM statement 4-6
  OPT statement 4-7
  PAG(E) statement 4-8
  REPT/ENDR statement 4-9
  SET statement 1-5,4-2
    function 4-2
    syntax 4-2
  SPC statement 4-8
  TTL statement 4-6
  USE statement 4-11
Assembler features i
Assembly listing format 1-6
  blank lines 1-5
  comment field 1-6
  comment lines 1-5
  label field 1-4
  operand field 1-6
  operation field 1-5
Assembly mnemonics 1-10
  address mode syntax 1-10
  register mnemonics 1-10

B

Binary Expressions 1-8
Blank lines 1-5

C

Character Constant Expressions 1-8
Comment field 1-6
Comment lines 1-5
Command line processing 1-2

D

Data storage offset 7-3
Decimal expressions 1-7
DC.S statement 5-2
DO.S statement 5-4
DS statement 4-10

E

END statement 4-1
ENDM 2-2
ENDS 3-3
EQU statement 1-5,4-2
  function 4-2
  syntax 4-2
Error codes Appendix B
Expressions 1-7
  binary 1-8
  character constants 1-8
  decimal 1-7
  evaluation conventions 1-7, 1-8
  external symbols 1-9
  hexadecimal 1-7
  operator evaluation 1-8
  symbolic names 1-8,1-9

F

FAIL statement 4-3

G

Global labels 1-5
Global variable storage 3-3

H

Hexadecimal expressions 1-7

I

IF/ELSE/ENDC statement 4-4
    IFEQ 4-5
    IFGE 4-5
    IFGT 4-5
    IFLE 4-5
    IFLT 4-5
    IFNE 4-5
    IFP1 4-5
Input file format 1-4
    blank lines 1-5
    comment fields 1-6
    comment lines 1-5
    label fields 1-5
    operand field 1-6
    operation field 1-6
Installation ii
Instruction mnemonics 1-10

J

Jump table 7-5

L

L68 (the linker)
    command line execution 6-2
    execution 6-2
    explanation of use 1-1,6-1
    installation ii
    linking non-OS-9 systems 6-5
    options 6-3,6-4
Label field 1-5
Labels 1-5
    exceptions (SET, EQU) 1-5
    global 1-5
    name conventions 1-5
Local variable storage 3-3
Location counters 3-5
LO.S statement 5-4

M

Macros
    Arguments 2-3
        \Ln 2-4
        # 2-4
    definition 2-1
    ENDM 2-2
    format 2-2
    internal labels (\@) 2-5
    MACRO 2-2
    nested 2-2
Mainline segment 3-5
Memory modules 3-1,3-2,7-1

N

NAM statement 4-6

INDEX

O

Object code generation 1-6
Operands 1-7
  expressions 1-7
    binary 1-8
    character constants 1-8
    decimal 1-7
    evaluation conventions 1-7,1-8
    external symbols 1-9
    hexadecimal 1-7
    operator evaluation 1-8
    symbolic names 1-8,1-9
Operation field 1-5
  instruction mnemonics 1-10
  object code generation 1-6
Operator evaluation 1-8
OPT statement 4-7
ORG statement 5-4
OS9 statement 5-5

P

PAG(E) statement 4-8
Process data area 3-1,3-2
Program development 1-2
Program example Appendix A,3-4
Program sections 3-1,3-3
  PSECT 3-1,3-3
  VSECT 3-1,3-3
Programming Rules 7-1
  memory module format 7-1
  non self modifying code 7-1
  position independent object code 7-2,7-4
  position independent data storage 7-2
PSECT 3-1,3-3,3-6
  definition 3-6
  parameters 3-6,3-7
    attrrev 3-7
    edition 3-7
    entry point 3-7

P (continued)

PSECT (continued)
  parameters (continued)
    name 3-6
    stacksize 3-7
    trapent 3-7
    typelang 3-6
  statements (list of) 3-6
  syntax 3-6
Pseudo-instructions
  ALIGN statement 5-1
  DC.S statement 5-2
  definition 5-1
  DO.S statement 5-4
  DZ.S statement 5-3
  LO.S statement 5-4
  ORG statement 5-4
  OS9 statement 5-5
  TCALL statement 5-6

R

R68 (the assembler)
  explanation 1-1
  execution 1-2
  installation ii
  options 1-3
Rdump (rdump68) 1-1
  options 1-1
Referencing data areas 7-3
  data area offset 7-3
Referencing program areas 7-4
Register mnemonics 1-10
Relocatable object files 3-1
  Format C-1
    external definition section C-3
    external reference section C-4
    header section C-1
    local reference section C-4
REPT/ENDR statement 4-9

S

SET statement 1-5,4-2
    definition 4-2
    syntax 4-2
SPC statement 4-8
Symbolic names in expressions 1-8,1-9
System calls (OS9 statement) 5-5

T

TCALL statement 5-6
TTL statement 4-6

U-Z

USE statement 4-11

Variable declaration 3-3
Variable storage declaration 3-3
VSECT 3-1,3-3,3-8
    definition 3-8
    statements 3-8
    syntax 3-8

MICROWARE SYSTEMS CORPORATION

DES MOINES